Compiling for Coarse-Grained Reconfigurable Architectures based on dataflow execution paradigm

A Thesis
Submitted for the Degree of
Doctor of Philosophy
in the Faculty of Engineering

by
Mythri Alle

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA
DECEMBER 2012
To My Family
Acknowledgments

I acknowledge my research supervisor Prof. S K Nandy for his support throughout the course of the thesis. He introduced me to the art of technical writing and helped me improve my writing abilities with utmost patience. I remember him sitting next to me and correcting portions of the paper in the process of teaching me. I specially thank him for the freedom and encouragement he has given me from the very beginning of my work. My sincere thanks to Dr. Ranjani Narayan with whom I had many fruitful discussions that shaped the work I carried out during my PhD. She taught me the importance of presenting the work and she had tremendous patience while correcting my presentations for even a small talk.

Many thanks to my colleague, Keshavan Varadarajan who was always ready to help and more than eager for any discussion. He used to go out of the way to accommodate any request from anybody in the lab. He happily took up my responsibilities whenever I was away. I owe a special thanks to him for the numerous reviews of my thesis without which my thesis would not be in the present form. I also thank Ganesh Garga who was the first to test the limits of the compiler and his analysis helped a lot in improving the compiler. Alexander Fell, Saptarsi Das, Prasenjit Biswas and Rajdeep Mondal were key people behind the architecture and without their help, it would have been impossible to debug the compiler. A special thanks to Dr. Balakrishnan Srinivasan whose insightful comments and questions helped me to think critically and improve my work. I thank reviewers of my thesis, whose comments improved the quality of the thesis significantly.

I also thank Sanjay, Sravanthi, Mitsu, Adarsh, and all other people in CAD Lab who made my stay at the lab lively. I will miss all the fun we used to have during the parties organized by Sanjay and Sravanthi. My friends Rahul and Vasanta were always there whenever I needed them. All these people made my stay at IISc memorable. Sincere thanks to Mallika who helped me through all the administrative process. I also thank Aswath, Eswar, and Somesh for helping me during my stay at CAD Lab.

I express my gratitude to all my teachers through my primary education until Master’s degree who imparted knowledge and made me capable of pursuing a doctoral degree. A special thanks to Laskhmi Narasimha teacher who encouraged me at a very early stage to pursue my interests. I also thank
Prof. Y N Srikant and Prof Priti Shankar who taught me basics of compilers.
I am grateful to my Parents and my sister, Deepthi for their unconditional love and support. My parents always encouraged me to do whatever interests me and without their support, I would not have been the same. I also thank my husband, Ramakanth for being more than understanding. He used to calm me down whenever I went into panic mode. Without his understanding and encouragement, this thesis would have been impossible. A special note to my son, Parthu whose smile used to refuel me with all the energy. I am lucky to have such a wonderful family and I dedicate this thesis to my family for all their sacrifices, support and encouragement.
Abstract

Coarse-Grained Reconfigurable Architectures (CGRAs) can be employed for accelerating computational workloads that demand both flexibility and performance. CGRAs comprise a set of computation elements interconnected using a network and this interconnection of computation elements is referred to as a reconfigurable fabric. The size of application that can be accommodated on the reconfigurable fabric is limited by the size of instruction buffers associated with each Compute element. When an application cannot be accommodated entirely, application is partitioned such that each of these partitions can be executed on the reconfigurable fabric. These partitions are scheduled by an orchestrator. The orchestrator employs dynamic dataflow execution paradigm. Dynamic dataflow execution paradigm has inherent support for synchronization and helps in exploitation of parallelism that exists across application partitions. In this thesis, we present a compiler that targets such CGRAs.

The compiler presented in this thesis is capable of accepting applications specified in C89 standard. To enable architectural design space exploration, the compiler is designed such that it can be customized for several instances of CGRAs employing dataflow execution paradigm at the orchestrator. This can be achieved by specifying the appropriate configuration parameters to the compiler. The focus of this thesis is to provide efficient support for various kinds of parallelism while ensuring correctness. The compiler is designed to support fine-grained task level parallelism that exists across iterations of loops and function calls. Additionally, compiler can also support pipeline parallelism, where a loop is split into multiple stages that execute in a pipelined manner.

The prototype compiler, which targets multiple instances of a CGRA, is demonstrated in this thesis. We used this compiler to target multiple variants of CGRAs employing dataflow execution paradigm. We varied the reconfigurable fabric, orchestration mechanism employed, size of instruction buffers. We also choose applications from two different domains viz. cryptography and linear algebra. The execution time of the CGRA (the best among all instances) is compared against an Intel Quad core processor. Cryptography applications show a performance improvement ranging from more than one order of magnitude to close to two orders of magnitude. These applications
have large amounts of ILP and our compiler could successfully expose the ILP available in these applications. Further, the domain customization also played an important role in achieving good performance. We employed two custom functional units for accelerating Cryptography applications and compiler could efficiently use them. In linear algebra kernels we observe multiple iterations of the loop executing in parallel, effectively exploiting loop-level parallelism at runtime. In spite of this we notice close to an order of magnitude performance degradation. The reason for this degradation can be attributed to the use of non-pipelined floating point units, and the delays involved in accessing memory. Pipeline parallelism was demonstrated using this compiler for FFT and QR factorization. Thus, the compiler is capable of efficiently supporting different kinds of parallelism and can support complete C89 standard. Further, the compiler can also support different instances of CGRAs employing dataflow execution paradigm.
## Contents

**Acknowledgments** ........................................ i

**Abstract** .................................................. iii

**List of Figures** .......................................... ix

**List of Tables** ........................................... xiii

**List of Algorithms** ........................................ xv

1 **Introduction** ........................................... 1
   1.1 What is a CGRA? ........................................ 2
   1.2 Compiling for a CGRA .................................. 3
   1.3 Motivation for a new compiler ......................... 5
   1.4 Overview of the work .................................. 7
   1.5 Contributions of the thesis ........................... 8
   1.6 Overall Setting of this Thesis ....................... 10
   1.7 Thesis Organization .................................... 10

2 **Design of the Compiler** ................................ 13
   2.1 Architectural parameters of a CGRA .................. 13
   2.2 Design of the compiler ................................ 15
   2.3 Target Architecture .................................... 18
      2.3.1 Synchronization Mechanism ........................ 18
      2.3.2 Model of Orchestration ............................. 20
   2.4 Compiler Design Considerations ...................... 21
      2.4.1 Imperative language to Dataflow execution ....... 21
      2.4.2 Dataflow Execution on a CGRA ................... 28
   2.5 Summary .................................................. 31

3 **Related Work** ........................................... 33
   3.1 PACT-XPP ................................................. 33
   3.2 RAW ....................................................... 35
   3.3 Montium .................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Ambric</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>ADRES</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>PPA</td>
<td>39</td>
</tr>
<tr>
<td>8.7</td>
<td>Black diamond</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Summary</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Compilation flow</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of compilation</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Dataflow Graph</td>
<td>45</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Memory Ordering</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Predicates</td>
<td>50</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Steer Nodes</td>
<td>52</td>
</tr>
<tr>
<td>4.2.4</td>
<td>ROBDD Optimization</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>HyperOps</td>
<td>58</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Overview of HyperOp</td>
<td>59</td>
</tr>
<tr>
<td>4.3.2</td>
<td>An algorithm to form HyperOps</td>
<td>62</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Correctness of the proposed algorithm</td>
<td>65</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Complexity analysis of the algorithm</td>
<td>69</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Measuring the effectiveness of Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Target Specific Customizations</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Resource Management at the Orchestrator</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of orchestrator</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>Compiler directed resource management</td>
<td>78</td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>Computing Purge Points</td>
<td>79</td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>Computing the number of inputs</td>
<td>80</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Synchronizing HyperOps</td>
<td>84</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Context Memory Allocation</td>
<td>86</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Delivering data to HyperOps</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>Hardware controlled resource management</td>
<td>89</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Data Delivery to a HyperOp</td>
<td>89</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Handling Loop invariant Data</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>93</td>
</tr>
<tr>
<td>6</td>
<td>Enhancements for performance</td>
<td>95</td>
</tr>
<tr>
<td>6.1</td>
<td>Executing Loops on Fabric</td>
<td>96</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Well behaved loops and Transforming to well behaved loops</td>
<td>97</td>
</tr>
<tr>
<td>6.1.2</td>
<td>End-to-end flow control mechanism</td>
<td>99</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Near-neighbor flow control mechanism</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>Other Enhancements</td>
<td>105</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Pipelining Loops on Fabric</td>
<td>105</td>
</tr>
</tbody>
</table>
List of Figures

2.1 High level View of a Reconfigurable System ............... 14
2.2 Parameters and their classification .......................... 16
2.3 An example showing a loop that does not require flow control
   mechanism for correct execution on fabric .................. 19
2.4 Abstract Model as seen by the compiler .................... 21
2.5 An example of transforming a code fragment into a dataflow
   graph (a) shows the code fragment that is being transformed
   to dataflow graph (b) Dataflow graph with only data depend-
   encies (c) dataflow graph after adding precedence edges ... 23
2.6 An example showing conditional operator implemented using
   switch nodes (a)Code Fragment (b) Dataflow graph imple-
   mented using switch nodes .................................... 24
2.7 Example of nested predicates (a) Code Fragment (b) shows
   adding predicates by compounding the predicates (c) Adding
   nested predicates ................................................. 25
2.8 Example of a loop implemented using switch nodes .......... 25
2.9 Reuse of context memory locations. (a) shows the case D is
   on taken path where it is safe reuse the context allocated to D
   (b) shows the case when D is on not-taken path and it is not
   safe to reuse the context allocated to D ...................... 31
4.1 A high-level overview of compilation flow .................. 44
4.2 Various steps in converting an application specification in
   imperative language into a dataflow graph ................... 46
4.3 Example showing addition of memory dependencies (a) shows
   a BB with a memory chain. For finding the appropriate memory
   chains that need to be linked, every memory chain is replaced
   by instruction to read followed by instruction write as shown in
   (b). (c) shows a control flow graph after performing Static Single
   Assignment (SSA) translation. BB B3 has a φ node, but does not
   have a memory operation. Instead of adding a dummy operation
   φ nodes are iteratively replaced (d) shows links added from all
   predecessors after iterative replacement ....................... 49
4.4 Example showing the latency between dependent memory operations. In this figure each box represents a computation element. Box marked with LSU represents the load-store unit. (a) shows the latencies when precedence edges are serviced at the Load/Store Unit (LSU). The memory trigger is delivered from the LSU. Hence, there is a round-trip latency from producer to consumer. (b) Shows the latency of the operation when all memory operations in an alias set are assigned to the same computation element and hardware ensures in-order delivery between any pair of communication entities.

4.5 Example showing the synchronizing memory operations. (a) shows the memory operations with memory precedence edges added. (b) shows the dataflow graph after adding the synchronizing memory operations. In both (a) and (b) only a part of the dataflow graph containing memory operations is shown.

4.6 Example showing predicate hoisting. (a) shows a control flow graph (b) shows a Data-Flow Graph (DFG) after the control dependencies are added. n1, n2, n3 are nodes in basic blocks B4, B5, B3 respectively. (c) displays the DFG after predicate hoisting is performed. The ROBDD of the CFG will be node B1 hence only one control dependency is added.

4.7 Example showing Optimization on steer nodes. (a) shows a control flow graph with partial instructions in basic block. (b) shows a DFG after the Steer nodes are added. n1, n2, n3 are nodes in basic blocks B4, B5, B3 respectively that deliver data to the steer node. (c) displays the DFG after ROBDD optimization is performed on steer nodes.

4.8 Example showing optimization of steer nodes. (a) shows a control flow graph and instructions in basic block. (b) displays a DFG after “steer” nodes are added. Since a3 can receive data from two possible control paths, two steer nodes are inserted. The dotted edges represent predicate edges. There is unconditional branch from basic block B2 to B3, hence steer node s2 is not predicated. Steer node s1 is predicated using b. (c) shows the DFG after optimizations are performed. Steer node s2 is not predicated and hence steer node s2 is eliminated. Steer node s1 is the only consumer of its parent, a1, so steer node s1 is eliminated.

4.9 Example showing Reduced Order Binary Decision Diagram (ROBDD) optimization. (a) shows control flow graph rooted at A which is immediate dominator of H (b) shows the corresponding Binary Decision Diagram (BDD) and also the logic expression for execution of BB H. We use a to represent a true path from A and a to represent a false path from A. and (c) shows the ROBDD.
LIST OF FIGURES

4.10 Plot showing the percentage reduction in non-compute nodes using ROBDD optimization. ............................................. 58
4.11 (a) shows the required partitioning of the Control Flow Graph (CFG) where Basicblock D is included in two HyperOps. (b) shows the HyperOps formed after duplicating Basicblock D so that HyperOps do not overlap ............................................. 60
4.12 A HyperOp that contains basicblocks from mutually exclusive paths ................................................................. 61
4.13 (a) Example showing the order in which basic blocks are included. We go into depth first and include basicblocks B, D, E, H before we include basicblock C as indicated by the dashed line. (b) shows the candidate basic blocks for the partially formed HyperOp and HyperOp cannot grow because of structural constraints (c) shows the case where a HyperOp cannot be increased because of mutually exclusive paths (d) shows the case where a HyperOp cannot include more basicblocks because of the absence of immediate dominator. ............. 64
4.14 Example showing the partitioning of dataflow graph .............. 65
4.15 Example control-flow graph to illustrate the computation of purge instructions ............................................................ 72
5.1 Example showing a CFG and HyperOp Interaction Graph ....... 81
5.2 Example showing chaining of acknowledgements when path $H1 \rightarrow H5$ is taken ............................................................. 86
5.3 Example showing different kinds of HyperOps .................... 88
5.4 Code sequence in figure (a) is an example of a nested loop along with the grouping of instructions into HyperOps. As shown in (b) HyperOp A and HyperOp B have only one instance, hence a static tag is sufficient. HyperOp C consumes data produced by HyperOp A multiple times and hence a sticky token is required. HyperOp C and HyperOp D have multiple instances hence a dynamic tag is required. A predicated steer node is necessary for communication between HyperOp D and HyperOp E, since HyperOp D produces the token multiple times. Only the last value should be given as input to HyperOp E. ........................................................................ 91
5.5 In the example shown, HyperOp A and HyperOp C communicate, but HyperOp A is not in the immediate parent loop and hence an additional node is added in HyperOp B to ensure each HyperOp communicates only with HyperOps in their parent loop. ........................................................................ 93
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>(a) Example of a loop that is not well behaved (b) Transforming the loop to well behaved loop by adding dependencies (c) Partitioning a non-well behaved loop such that no flow control is required for its execution</td>
</tr>
<tr>
<td>6.2</td>
<td>Different types of pHyperOp interaction graph</td>
</tr>
<tr>
<td>6.3</td>
<td>An example illustrating the flow control mechanism implemented per loop using a leader</td>
</tr>
<tr>
<td>6.4</td>
<td>An example code sequence and one possible mapping of the code onto the fabric. With the mapping shown in this figure it leads to an incorrect execution (a) shows the snapshot when the CE D has finished the first iteration (b) shows the snapshot when the data is delivered to CE C from CE B for the first iteration. By this time, CE D would have finished second iteration. (c) shows the snapshot when the third iteration data from CE B is ready at CE C, but second iteration data is yet to be received</td>
</tr>
<tr>
<td>6.5</td>
<td>An example mapping of the code onto fabric such that every pair of communicating entities transfer data for every iteration of the loop</td>
</tr>
<tr>
<td>7.1</td>
<td>Block Diagram showing the interconnection between the reconfigurable fabric and the orchestrator</td>
</tr>
<tr>
<td>7.2</td>
<td>Block diagram showing the pipeline stages of the pipeline</td>
</tr>
<tr>
<td>7.3</td>
<td>Graph showing the control overhead for various applications</td>
</tr>
<tr>
<td>7.4</td>
<td>Graph showing the break-up of different kinds of edges in the dataflow graph</td>
</tr>
<tr>
<td>7.5</td>
<td>Plot showing the number destinations in the dataflow graph</td>
</tr>
<tr>
<td>7.6</td>
<td>Plot showing break-up of different kind of nodes after forming HyperOps</td>
</tr>
<tr>
<td>7.7</td>
<td>Plot showing the break-up of different kinds of edges after forming HyperOps</td>
</tr>
<tr>
<td>7.8</td>
<td>Execution time of different platform variants normalized w.r.t. to Configuration I</td>
</tr>
<tr>
<td>7.9</td>
<td>Execution time of our platform when compared with other accelerator approaches</td>
</tr>
<tr>
<td>7.10</td>
<td>Execution time of our platform when compared with a General purpose processor</td>
</tr>
<tr>
<td>7.11</td>
<td>Execution time of our platform when compared with a General purpose processor</td>
</tr>
<tr>
<td>7.12</td>
<td>Execution time of our platform when compared with a General purpose processor</td>
</tr>
</tbody>
</table>
List of Tables

7.1 List of all Function Units (FUs) and the latency in clock cycles for the Cryptography fabric and floating-point fabric. . . . . . . 113
7.2 Details of the two platform variants used in our experiments . 114
7.3 List of experiments . . . . . . . . . . . . . . . . . . . . . . . . 122
List of Algorithms

1. An Algorithm to create HyperOp ........................................... 66
2. Pseudo-code of a function to update the active list ................. 67
3. Pseudo-code of a function to choose a basic block for inclusion
   into a HyperOp .............................................................. 67
4. An Algorithm to partition the basicblock into multiple HyperOps 68
5. An algorithm to compute the set of operations to be purged at
   each branch point ......................................................... 74
6. Pseudo code showing the computation of aggregate number of
   inputs received for a HyperOp, at each HyperOp ................... 82

ALU Arithmetic Logic Unit
ASIC Application Specific Integrated Circuit
CDFG Control-Dataflow Graph
DFG Data-Flow Graph
CGRA Coarse-Grained Reconfigurable Architecture
DLP Data Level Parallelism
FU Function Unit
HDL Hardware Description Language
HLL High-level Languages
ILP Instruction Level Parallelism
I/O Input/Output
NoC Network-on-Chip
RISC Reduced Instruction Set Computer
CFG Control Flow Graph
SSA Static Single Assignment
LLVM  Low Level Virtual Machine
TLP  Thread/Task Level Parallelism
VLIW  Very Long Instruction Word
CRC  Cyclic Redundancy Check
IDCT  Inverse Discrete cosine transform
FFT  Fast Fourier Transform
SHA-1  Secure Hashing Algorithm-1
BDD  Binary Decision Diagram
ROBDD  Reduced Order Binary Decision Diagram
LSU  Load/Store Unit
FDS  Force Directed Scheduling
AES  Advanced Encryption Standard
AES-E  Advanced Encryption Standard Encryption
AES-D  Advanced Encryption Standard Decryption
ECPA  Elliptic Curve Point Addition
ECPD  Elliptic Curve Point Doubling
LICM  Loop-Invariant Code Motion
MRI-Q  Magnetic Resonance Imaging-Q
GPP  General Purpose Processor
Coarse-Grained Reconfigurable Architectures (CGRAs) consist of interconnection of computation elements and this interconnection of computation elements is referred to as reconfigurable fabric. These computation elements operate at the level of words as opposed to fine-grained reconfigurable architectures that operate at bit-level granularity. The reconfigurable fabric can be configured to do a particular task by loading appropriate configuration information. The configuration information is generated by the compiler by suitably mapping the application specification onto the reconfigurable fabric. The process of generating this configuration information is comparable to that of placement and routing step employed in hardware synthesis process. Several techniques from the domain of hardware synthesis have been adopted for mapping the applications onto the CGRA (Friedman et al., 2009; Hiroyuki et al., 2006). When an application cannot be accommodated on the reconfigurable fabric, the application needs to be partitioned such that each application partition can be executed on the fabric. Further, we need a mechanism to enable data transfer across these partitions and schedule these partitions. A special purpose unit can be employed to improve the efficiency of such a mechanism. We refer to such a hardware unit as an orchestrator. Orchestrator employing dataflow execution paradigm helps us exploit fine-grained Thread/Task Level Parallelism (TLP) that exists across function calls. It also allows us to exploit loop-level parallelism. Compiling for such a CGRA involves partitioning the application and providing necessary support for orchestrating these partitions in addition to mapping of these partitions onto the reconfigurable fabric. In this thesis, we look at the first two aspects of compiling onto a CGRA employing dataflow execution paradigm at the orchestrator. Further, the compiler is designed to target multiple architectural variants of the reconfigurable fabric. The mapping process of application partitions depends on the particular instance of the fabric. We do not delve
into this aspect of compilation in this thesis. The focus of this thesis is on correctness aspects related to partitioning the application and orchestrating these application partitions.

In this chapter, we provide an overview of CGRAs. This is followed by the compilation aspects of the CGRA. We also motivate the need for a new compiler for a CGRA. We conclude the chapter with a description of the organization of the thesis.

1.1 What is a CGRA?

CGRAs have evolved over the past two decades and there is no single definition that is accepted by everyone. At a high-level, a CGRA is an interconnection of computation elements. These computation elements interact directly through the network instead of employing a shared memory as in multi-core architectures. This interconnection of computation elements is referred to as reconfigurable fabric. The reconfigurable fabric is coupled with either a host processor or a dedicated orchestrator to manage the fabric. Several variants of CGRAs with different choices of computation elements, interconnection and coupling with host processor have been explored. Early CGRAs typically consisted of FUs and employed a programmable interconnect. Kres-Array (Hartenstein et al., 1998) and DP-FPGA (Cherepacha and Lewis, 1996) are examples of such systems. These are FPGA-like CGRAs at a higher level of granularity. Though these CGRAs had lesser overhead in terms of configuration time when compared to that of FPGA, it was still significant. To alleviate this problem multiple configuration planes were provided and the switching of configuration across these planes was in the order of few cycles. Each configuration plane includes the configuration for all the computation elements and switches on the fabric. The configuration plane is a global unit and switching between configuration planes, affects all the computation elements on the fabric. In other words, computation elements execute in lock step. These reconfigurable systems were coupled with the host-processor and are used as a reconfigurable FU or as a co-processor for accelerating specific kernels viz. loop bodies or function calls. In both cases, host-processor was responsible for loading the correct configuration data and activating it when necessary.

In more recent CGRAs, the configuration planes have been replaced by instruction buffers and FUs by Arithmetic Logic Units (ALUs). Though the functionality of configuration planes and instruction buffers are the same, they indicate a change in the way CGRAs are programmed. The instruction buffers are placed along with the computation element as opposed to a global configuration plane. Since the instruction buffers are local to computation elements, this allows computation elements to work independent of other computation elements. Along with the instruction buffer, they also consist of
register files that are local to the computation element. These computation elements are typically interconnected using either point-to-point or program-

mable interconnect. The computation element employed varies from an ALU to a simple Reduced Instruction Set Computer (RISC) core. In ADRES (Mei et al., 2003a) and PPA (Park et al., 2009) the computation element consists of ALU or a domain specific functional unit. RAW (Taylor et al., 2002) employs a simple RISC core, MIPS R2000, as a computation element. The RISC core is replete with instruction, data caches and register files. Unlike most of the earlier reconfigurable systems, RAW allows computation elements to work independent of each other. RAW architecture is as close to tiled processors as they are to reconfigurable architectures. We classify RAW as a reconfigurable architecture due to the explicit communication required between the processing cores. RAW also provides two layers of network viz. static network and a dynamic network. Static network is programmed by the compiler to exchange data when the dependency can be determined at compile time. Dynamic network is provided to handle situations that cannot be determined at compile time. In both cases, the transports are explicitly specified and do not go through shared memory as it happens in multi-core/tiled architectures.

Thus, CGRAs are different from other distributed architectures exploiting spatial computation in the following aspects.

• The communication between the computation elements are explicitly specified in a CGRA. These communications are specified using explicit instructions. Though computation elements can also communicate using memory, the primary mode of communication is using the network available on the fabric.

• In a CGRA, computation elements have shorter communication latencies when compared to processors communicating with each other. Therefore, we can afford to exploit fine-grained parallelism when compared to threads in a GPP. The fine-grained parallelism can exist across various iterations of the loop or across two function calls within an application.

• The instruction buffers of CGRA are exposed to the compiler. These are similar to zero overhead loop buffers in DSP processors. This allows compiler to schedule the instructions efficiently and thus achieve higher performance.

1.2 Compiling for a CGRA

CGRAs are designed to accelerate computational workloads that operate at the granularity of words (integers/floating point numbers). Hence, they are
programmed using High-level Languages (HLLs) as opposed to Hardware Description Languages (HDLs) used for programming fine-grained reconfigurable architectures. HDLs are designed to work at the granularity of bits and require description at register-transfer level. Hence, they are not suitable for programming a CGRA. Several research groups have proposed the use of custom languages specific for their CGRA. For example, Dataflow Intermediate Language (DIL) is a language used to program PipeRench (Goldstein et al., 2000). Few frameworks extend an existing high-level language by adding support to specify the concurrency and synchronization. For example, RapiD-C (Cronquist et al., 1998) is an extension of C language with support for specifying parallelism, partitioning, and data transfers explicitly. Others accept a high-level language directly without adding any custom extensions. Examples of these include ADRES, PPA that accept an application specification in C language, and RAW that accepts FORTRAN or C. Use of such high-level imperative languages makes the task of compilation more complex. Imperative languages are inherently sequential and determining parallelism from such a specification is a challenge in itself. Though there has been some research for finding a suitable programming paradigm for a CGRA, there is no single solution that is adequate for all situations. However, from the earlier attempts it is clear that, custom languages and custom extensions limit the acceptance of the CGRA since it involves a learning curve to program the CGRA. An ideal solution for this problem is to define a specification that can be used across all CGRAs. There has been little research in this direction. We believe that an existing language with support for intrinsics or other minor extensions is the most suitable form to specify applications when targeting a CGRA. In this thesis, we present a compiler that accepts an application specified in C language. This language specification can be supplemented through use of intrinsics, user hints etc.

Compiling for early CGRAs is akin to the process of synthesizing a netlist. It involves programming the computation elements and determining the paths for exchanging data between them. These tasks are similar to that of placement and routing step of the synthesis process. In CGRAs that support multiple configuration planes, in addition to the synthesis, it also involves folding the input specification onto the same set of computation elements when a single configuration plane is not sufficient to accommodate the entire specification. The number of different configurations that are allowed for a computation element depend on the number of configuration planes available. This step introduces more constraints on the synthesis of applications, as the data from an earlier configuration has to be transferred to the current configuration. In recent CGRA, the presence of instruction buffers and the register file within a computation element pushes the process of programming a CGRA farther away from the traditional synthesis process. In such kind of architectures, the mapping of applications can be compared to that of compiling for clustered VLIW processors. However unlike VLIW
1.3 Motivation for a new compiler

Compilers built for CGRAs thus far have been specific for the target it was built for. This field is yet to see the maturity seen in compiler for general-purpose processors. Some of the reasons for this include (i) use of new or domain specific languages (ii) use of language extensions which are very specific for the CGRA. Hence, we use a C language to program the CGRA. We support full C89 standard.

Most of these CGRAs are tightly coupled with a host processor and hence the focus has always been mapping application partitions onto the CGRA. The orchestration of these partitions is responsibility of the host processor. Since communication with host processor can involve significant delays, such architecture may not be very efficient in exploiting parallelism across partitions. As the complexity of applications increase, accelerating only parts of the application is not sufficient to achieve necessary performance. According to Amdahl’s law, the speed-up achieved is limited by the fraction of the time spent in the accelerated portions. To accelerate the entire application, it is essential to employ an efficient mechanism to partition the applications

---

For example, gcc is a standard compiler shared across multiple architectures
and orchestrate these partitions. The application should be partitioned such that each application partition can execute on the fabric without requiring a context switch. Since the context of the application is distributed among multiple computation elements on the fabric, it is not efficient to context switch an application partition. Further, the orchestrator should be capable of exploiting the fine-grained parallelism that exists across the iterations of the loop or across the function calls in an application. Along with this fine-grained TLP it should also be capable of exploiting other kinds of parallelism viz. Instruction Level Parallelism (ILP), Data Level Parallelism (DLP), and pipeline parallelism efficiently.

In this thesis, we present a compiler that supports different kinds of parallelism viz. TLP, DLP, and pipeline parallelism through correct and efficient orchestration of application partitions. This thesis focuses on the correctness aspects of the orchestration and expects the user to indicate the parallelism available explicitly. Several techniques like polyhedral analysis and inter-procedural analysis can be employed to automate this task. Further as suggested by a study by Borkar and Chien (2011), domain customization is essential to achieve high performance and energy efficiency. Hence, it is essential for both the architecture and compiler to have capability to support domain customizations. Several compilers support a sub-set of these features. Several compilers viz. DRESC (Mei et al., 2003a), PPA Park et al. (2009), Molen Vassiliadis et al. (2001) are designed to accelerate portions of applications. XPP-VC (Baumgarten et al., 2003), RAWCC (Taylor et al., 2002) and Montium Compiler (Heysters et al., 2003) are designed to accelerate entire applications. However, both XPP-VC and Montium lack the support for task-level parallelism. RAW is designed as a general purpose machine and support for domain customization and pipeline parallelism is not available. The objective of this work is to build a compiler that can support all the features mentioned earlier. Another objective of this work is to provide one compiler to target a family of architectures. Several parameters viz. size of instruction buffer, type of interconnection should be customized for a set of applications. A parameterized compiler that can target different variants of a CGRA is necessary to ease the design space exploration to arrive at the most suitable architecture for a set of applications. To be able to do so, we first present the abstract model of the orchestrator to which the compiler is targeted. The abstract model is chosen to exploit TLP and other forms of parallelism efficiently. This is also designed with capabilities to execute the entire application, reducing the burden on the host processor. In summary, we aim at building a compiler that is capable of

- Supporting complete C specification without the need of any additional constructs. Intrinsics should be supported to achieve higher perform-
1.4 Overview of the work

In this thesis, we present a compiler that can compile for a class of CGRAs that employ a dataflow execution paradigm. The first step of the compiling onto such CGRAs is to bridge the gap between imperative specification and dataflow execution by constructing an efficient dataflow graph. Then, the dataflow graph is partitioned such that each partition can be executed on the reconfigurable fabric without need for context switching. Since the context of the application partition is distributed across several computation elements, context switching is difficult to implement. To increase the resource utilization, an application partition is launched onto the reconfigurable fabric only when the data required for the partition is available. We derive necessary and sufficient conditions for the correct execution of these partitions under the above mentioned constraints. The data required for an application partition is accumulated by the orchestrator and it schedules the partition once all the data for the partitions is available. Orchestrator maintains the data required for an application partition in a context memory location assigned for that partition. Additional information is required to distinguish data designated for multiple instances of the same partition. Tags are employed (akin to tagged-token dataflow execution paradigm) along with data to identify the correct instance of the destination. For applications with statically known dependencies, compiler can efficiently manage these context memory locations thereby reducing the complexity of the hardware. In this scheme, compiler has the responsibility of assigning context memory locations for each of the application partitions. We support both these orchestration mechanisms viz. compiler directed context memory management and hardware controlled context memory management. The compiler also provides support for do-
main customization, which is the key for achieving good performance and energy efficiency.

When compiling for dataflow architectures, switch nodes are employed to deliver data to an operation only when required. These nodes are only responsible for routing the data to an appropriate instruction based on the branch decision and do not perform any computation. The percentage of switch nodes can be as high as 45% for highly control driven applications. Hence, to reduce such kind of nodes, we deliver data directly and instead predicable the execution of operations. Since the data delivery is not predicated, data could be delivered for an application partition that is on a not-taken path. To reuse context memory locations assigned for such partitions compiler needs to introduce instructions for clearing unused data. Since the execution of the application partition is distributed and it can potentially introduce non-deterministic delays, the mechanism employed to clear the unused data should be capable of withstanding variable latencies. We present such a mechanism in this thesis.

To show the flexibility of the compiler, we use it to target multiple variants of CGRAs employing dataflow execution paradigm. The compiler was used to compile applications for architectures employing both modes of orchestration viz. compiler directed memory management and hardware controlled context memory management. Along with the orchestration mechanism, we have also varied the reconfigurable fabric employed. One of them contains a homogeneous fabric of ALUs interconnected with a NoC and the other employs a heterogeneous fabric consisting of floating point and integer units. We have also varied other parameters like the size of instruction buffer employed. Applications from two domains viz. cryptography, linear algebra are chosen to evaluate the compiler. Cryptography applications are targeted to CGRAs employing homogeneous fabric whereas linear algebra kernels were targeted to heterogeneous fabric. The fabric used for cryptography employed two custom function blocks along with the ALU. We observe performance improvement of over an order of magnitude to close to two orders of magnitude in the case of cryptography applications. In the case of linear algebra applications, we see significant performance degradation. According to our initial analysis, the performance degradation is due to the lack of suitable memory hierarchy and domain customization.

1.5 Contributions of the thesis

This thesis demonstrates a compiler that targets CGRAs employing dataflow execution paradigm. The compiler has been designed to exploit fine-grained task level parallelism that exists across loop iterations or function calls. Further, it can also support pipeline parallelism, which is an important feature for supporting streaming applications. Dataflow execution paradigm plays a
1.5 Contributions of the thesis

key-role in exploiting different kinds of parallelism. Thus, the important task of the compiler is to transform an imperative specification onto a dataflow architecture by exposing as much parallelism as possible. More specifically, the main contributions of this thesis include

- **ROBDD based optimization in the context of CFG to obtain efficient dataflow graphs.** It is known from the earlier literature that naive translation of a CFG into dataflow graph will incur a large overhead in terms of the number of non-compute nodes added to transform control flow into dataflow. The proposed optimization computes the minimal number of non-compute nodes required for such a transformation.

- **Deriving the necessary and sufficient set of constraints to ensure correct execution of the application partitions on a dataflow oriented CGRA.** We also propose an algorithm that uses information from CFG and DFG to automate the generation of application partitions from the given application specification.

- **Mechanism to tag the data tokens exchanged between multiple iterations of the loop without the need of a centralized entity to generate tokens.**

- **We also propose an efficient scheme to reclaim the context memory locations allocated to partitions on not-taken paths.** This is not required in traditional dataflow machines since switch nodes are employed to deliver data only when required. The percentage of switch nodes would be very high for control-intensive applications. Hence, we do not employ switch nodes and instead deliver data directly. Since the data is delivered directly, it is possible that partial data is available for partitions on not-taken paths. We propose a scheme to clear such data with minimal hardware support. This scheme also aids in defining the def and use of a partition which helps in assigning the context memory locations statically.

- **We present three different techniques to support execution of loops on the fabric that does not employ a flow-control mechanism.** Static dataflow execution paradigm supports such an execution by employing acknowledgements between every pair of communicating operations. However, such a scheme would result in as many acknowledgements as data tokens. Therefore, we do not employ acknowledgements to support loops. We present three different schemes each with different performance vs. hardware complexity tradeoffs.
1.6 Overall Setting of this Thesis

This thesis is a part of a larger project in which the CGRA\textsuperscript{1}, its compiler, partitioning and mapping algorithms, Network-on-Chip (NoC) were developed. This thesis pertains only to the compiler of the CGRA and its implementation. Other theses cover different aspects of this work. These include

1. The thesis by Varadarajan (2012) provides the details of the architecture.

2. The thesis by Krishnamoorthy (2010) provides the details of the partitioning and mapping algorithms that can be used along with this architecture.

3. NoC is detailed in the architecture by Fell (2012).

1.7 Thesis Organization

The objective of this thesis is to design a compiler targeted for a family of CGRAs and is capable of exploiting various kinds of parallelism. It should have support to take user specified compiler hints for performance critical code sequences. The compiler accepts a complete C standard and intrinsics are employed to improve the performance. In this thesis, we discuss the challenges in developing such a compiler and various techniques we proposed to address each of them. The thesis is organized as follows:

- Chapter 2 presents the abstract model of the architecture. This model acts as a bridge between the compiler and the architecture. Compiler is designed to target this architecture. The architecture follows a dataflow execution paradigm. It has an inherent support for lightweight synchronization and is suitable for exploiting TLP. This section also discusses issues involved in compiling for such architecture.

- In Chapter 3, we present related literature. We discuss the similarities and the differences of this compiler with other compilers from literature.

- Chapter 4 presents the core of the compilation flow. It presents two important phases of the compilation. In the first phase, the imperative specification is transformed to a DFG\textsuperscript{2}. We discuss the choices available for transforming imperative specification into a dataflow graph and discuss the suitability of these choices in the context of CGRAs. We also propose a ROBDD-based technique to eliminate the redundancies in the dataflow graph to make them more efficient. In this chapter, we
present an algorithm to partition the application into multiple partitions referred to as HyperOps. Here, we discuss various constraints imposed on these HyperOps to ensure correct execution. We also prove that these constraints are necessary and sufficient to ensure correctness.

- Chapter 5 delves into the aspects related to the orchestration of HyperOps. The compiler support for orchestrating the HyperOps depends on the orchestration mechanism employed in the hardware. As mentioned earlier, we follow a dataflow execution paradigm and support two variants of dataflow orchestrator. In the first variant, the resources at the orchestrator are managed by the hardware. In the second variant, the resources are exposed to the compiler and compiler manages the resources. The compiler support for both these schemes is discussed in this chapter.

- Chapter 6 looks at techniques proposed to enhance application performance. The major portion of this chapter focuses on support for executing loops on the reconfigurable fabric. In loops that lack loop-level parallelism, it is efficient to keep the loop resident on the fabric to avoid load latencies. Unlike the earlier attempts that assume deterministic latencies for mapping loops, we support variable latencies. We also discuss flow control mechanisms to support execution of nested loops on the fabric and to exploit pipeline parallelism. We also briefly mention the support for domain customization, TLP and pipeline parallelism.

- We evaluate the efficiency of the proposed approach using applications from two different domains. Chapter seven presents the results of this evaluation and also compares, the performance obtained with other related work and a General purpose processor. We also analyze the results and point to the bottlenecks in the current design.

- We conclude the thesis in Chapter 8. In this chapter, we summarize the work and highlight the contributions of this thesis. We then present few important observations that pave the way for future work.
Chapter 2

Design of the Compiler

Unlike traditional processors that change infrequently, CGRAs are customized for a domain/application by fine-tuning various parameters of the architecture. Further, a lot of design space exploration is performed before arriving at the right instantiation for a particular domain. The customizations include fine-tuning the numbers of computation unit, amount of instruction buffer available and the type of interconnection between computation elements etc. Another important factor in CGRAs is the lack of requirement for backward compatibility because of which the customizations/up-gradations are less constrained. This freedom poses challenges when designing a compiler since it should be capable of supporting a wide variety of platforms without having to re-engineer significant portions of the compiler. Further, the compiler should be customizable to efficiently support design space exploration. To design such a compiler, we first identify the set of architectural parameters that define a CGRA. We present the design of the compiler and the motivation behind such a design. We then discuss the design considerations for building a compiler for the chosen set of parameters.

2.1 Architectural parameters of a CGRA

As mentioned previously, a CGRA comprises an interconnection of computation elements. Each of these computation elements have instruction buffers, which are used for storing instructions to be executed by that computation element. These instructions are typically loaded by an external entity and are not fetched by the computation elements. Further, these computation elements communicate directly through the network provided. The total number of instruction buffers available on the reconfigurable fabric may be insufficient to accommodate all instructions of the application. When an entire application cannot be accommodated on the fabric, the application is
partitioned into multiple application partitions. The orchestration of these application partitions is ensured by an orchestrator. Figure 2.1 depicts a high-level overview of such a system.

![Figure 2.1: High level View of a Reconfigurable System](image)

The design space of a CGRA can be expressed in terms of three near-orthogonal parameters viz. choice of computation element, choice of interconnection network and choice of orchestrator. Each of these three parameters presents a number of sub-choices. These are listed below:

- **Computation Element:**
  - Mode of Instruction Selection: This determines the next operation to be executed. The choices include a static schedule as determined by the compiler, a schedule based on data availability, PC-based execution as implemented in von-Neumann machines. The difference between PC-based and the static schedule is that, in a static schedule all instructions are executed. Some of these might become NOPs depending on the control condition. There are several other variants available.
  - Choice of Synchronization Mechanism: Since the computation is distributed, a synchronization mechanism is required to ensure correctness. The synchronization mechanism determines (i) when the data is ready to be used by an operation (ii) when the data is ready to be sent to the destination. The synchronization mechanism employed depends on if the compute units work in lockstep or they can execute independently. When computation elements work in lockstep, compiler can statically determine schedules to enforce a synchronization mechanism. When the computation elements work independent of each other, handshake mechanism needs to be implemented across the compute units. We discuss more about these later in this section.
  - Set of operations supported in each computation element.
2.2 Design of the compiler

One of the objectives of designing this compiler is to provide efficient orchestration to exploit parallelism that is available across application partitions. To achieve this objective we employ a dynamic dataflow execution paradigm at the orchestrator. As we discuss later in this chapter this orchestration model provides support for lightweight synchronization making it most suitable form of orchestration in the context of CGRA. Another essential feature for a compiler that targets CGRA is the ability to support multiple variants of a CGRA. A compiler that can be customized to target several instantiations of a CGRA eases the architectural design space exploration. Hence, we designed

- Size of instruction buffer
- Number of data registers available
- Organization of the computation element viz. pipeline depth, Instruction buffer access latency, register access latency etc.

- interconnect
  - Type of interconnection. The choice of the interconnection includes programmable, point-to-point, or an NoC. A Hybrid of these interconnects also can be employed to interconnect the computation elements. Further, multiple networks can be employed to suit the demands of the application.
  - Flit size, which defines the size of data that is transferred on the network. This typically corresponds to the width of operations supported by the computation element.
  - Topology of the network employed. The most popular topology is mesh-based interconnect. However, there are several other choices viz. honeycomb, hexagonal.

- Orchestrator
  - Model of orchestration: The orchestrator is responsible for scheduling application partitions and facilitating data transfer across application partitions. The model of orchestration determines the mechanisms employed to perform these tasks. At a high level, the orchestration mechanism can be based on control flow execution paradigm or dataflow execution paradigm. In both these execution paradigms, the orchestrator can be an external entity or the computation elements can be programmed to act as an orchestrator.
  - Mechanism for launching configuration of application partitions.
the compiler so that it can accept several configurable parameters. Figure 2.2 shows these parameters classified into multiple levels of abstraction based on the ease with which these parameters can be customized to target other CGRAs.

![Diagram of parameters and their classification]

- **Level-1 Parameters**: Synchronization mechanism and Model of orchestration are fixed, and these parameters form the foundation of the compiler. Hence these parameters are difficult to customize and the compiler is designed to target several variants of CGRAs employing the specified synchronization mechanism and model of orchestration.

  Synchronization mechanism specifies how the availability of data for an instruction is detected. i.e. the necessary condition for operation execution. Further, it also determines when the data produced by an operation is safe to be transmitted. Thus, synchronization determines the handshake protocol across different computation elements for exchanging the data. The most often used synchronization mechanism in CGRAs is to assume a time-synchronous execution. In this mechanism, all computation elements work in lockstep and compiler schedules the operations statically after accounting for all the latencies. This is similar to the scheduling of operations in a VLIW machine with a large number of functional units. When a time-synchronous mechanism is employed, it is not possible to map applications that have variable latencies. To tolerate variable latencies, the availability of data can be detected at run time. A simple way to implement it is by employing valid bits to indicate availability of data. We discuss more about this subsequently.

\[1\] Though a mechanism similar to assumed latencies employed in Itanium can be used to tolerate variable latencies it is not a practical choice in a CGRA, since (i) the number of computation elements is large and (ii) CGRA is typically employed in embedded systems and it is required to keep the hardware simple.
Model of orchestration determines the interface between application partitions. It also specifies a mechanism to schedule them. As mentioned earlier, the application needs to be partitioned into application partitions to execute them on a CGRA. Since, the reconfigurable fabric can accommodate only a certain number of instructions, applications have to be partitioned such that each application partition can be executed on the fabric. The orchestration mechanism determines the kind of parallelism that can be exploited on the platform. In addition, the complexity of this scheme determines the granularity at which such a mechanism can be used. For example, if the orchestration mechanism introduces a delay of thousands of cycles then it is not practical to use this scheme for application partitions that execute for hundreds of cycles. The delays incurred because of this orchestration should be significantly lesser than the average execution time of the partition to achieve good performance. Thus, this mechanism determines the granularity of application partitions for an efficient implementation. This mechanism also determines the interface across the partitions and the role of compiler in supporting such communication. It is difficult to build a compiler that can target widely varying orchestration models and hence we classify it in this level. Since, these two parameters form the foundation of our compiler we discuss them in detail in a subsequent section.

- Level-2 Parameters: The type of interconnection network used and the instruction selection model are present at this level. These two parameters affect the mapping step of the compilation where the operations belonging to an application partition are assigned to computation elements on the fabric. The mapping step also has the responsibility of routing the data across operations depending on the interconnection network employed. This step determines how an application partition is executed on the fabric and does not affect the interface between application partitions. Though it is difficult to customize the same set of algorithms to suit different choices of the interconnection network, it is possible to design them as modules that can be invoked depending on the interconnection employed. To be able to explore various choices, we design the compilation infrastructure in a modular fashion. The mapping step is designed as pluggable modules to be able to explore various choices of interconnection network and instruction selection mechanism without affecting rest of the compilation trajectory. This module takes an application partition as an input and generates mapping based on the hardware description.

- Level-3 Parameters: The next level contains parameters such as number of registers, size of instruction buffer, topology of the network,
instruction set etc. Though these parameters are equally important to map applications, it is possible to develop algorithms that accept these as configuration parameters. For example, it is possible to design an efficient algorithm to map an application partition onto a set of computation elements, where the number of computation elements is a configurable parameter. Our compiler is built to accept these as configuration parameters that are determined through design space exploration. The algorithms employed in various phases of compilation are designed to accept these as configuration parameters. In the domain of CGRA, where customization is the key to achieve performance it is essential to have a compiler that is parameterized largely.

- The last level contains implementation details hidden from the compiler.

### 2.3 Target Architecture

As mentioned earlier, we design our compiler such that Level-2 and Level-3 parameters can be fine-tuned based on the requirements of the application. Level-3 parameters can be changed by changing the configuration parameters of the compiler and Level-2 parameters can be customized by providing appropriate modules to perform the mapping step of the compilation. In this section, we discuss Level-1 parameters in detail and motivate the choices we have made for these parameters. As mentioned previously, Level-1 parameters cannot be modified.

#### 2.3.1 Synchronization Mechanism

Synchronization is required (i) at the consuming operation to ensure correct data is available before performing the operation and (ii) at the producing operation so that data at the consumer is not overwritten before it is consumed. The computation elements employed in a typical CGRA are simple (they range from a simple ALU to a simple RISC processor) and hence a complex synchronization mechanism will outweigh the energy benefits obtained by exploiting parallelism. Synchronization at consumer requires information that is local to the consumer and hence is simple to implement. A valid bit can be employed along with each data token to implement such synchronization. Valid bit is set when the data is written into it and it is reset after it is consumed. When a data is consumed multiple times, data can be replicated such that each data token is consumed exactly once. Another way to handle it is to identify the last use of the data and clear the valid bit after the last use. We employ a mechanism similar to that of dataflow execution. In this
2.3 Target Architecture

execution paradigm, data is replicated and one data token is delivered to each consumer.

Synchronization at producer requires more complex logic since it requires information about the consumer at the producer computation element. This synchronization mechanism is nothing but a flow control mechanism that ensures the producer does not overrun the consumer. One way to implement flow-control is to send an acknowledgement back to the producer once consumer has consumed the data (akin to static data flow model proposed by Dennis (1980)). In such a model, the number of acknowledgements will be as many as the number of data items transferred. It is to be noted that, a flow control mechanism is required only when a location is reused. Further, if we can ensure that data has been consumed before new data is produced, flow control mechanism is not required. In dataflow execution paradigm, every instruction is assigned a unique location and reuse happens only at loop boundaries, where the location is reused by multiple iterations of the loop. In many of the applications, data dependencies ensure that a producer can never overwrite data at the consumer. For example, consider the dataflow graph shown in figure 2.3 operation C can fire only after operation A has consumed data. Hence, whenever operation C is ready to fire, it means that operation A is ready to accept data. This example produces correct results even in the absence of a flow control mechanism. Since flow control is not required often, providing for such a support in hardware might not be energy efficient. As we discuss later in section 6.1 efficient schemes can be implemented using flow control provided by interconnection network or by using other software mechanisms. However, for a particular set of applications it might be efficient to implement such a mechanism in hardware. The compiler itself should be capable of handling both situations. The execution model of a fabric is based on a dataflow model. The synchronization mechanism describes the readiness of the instruction to execute. Also, note that it does not specify the instruction selection mechanism.
2.3.2 Model of Orchestration

Orchestrator is responsible for facilitating data transfer across the partitions, and scheduling the partitions. In other words, Orchestrator sees these partitions as operations and it needs to orchestrate these operations similar to a traditional processor. Unlike simple operations that are typically monadic or dyadic, it is a macro operation and has a higher number of inputs. The number of inputs needs to be specified explicitly as they vary from one operation to another. As with the traditional processors there are two modes of orchestration viz., control flow based orchestration and dataflow based orchestration. In the context of CGRA, since the execution of macro operation is distributed and the presence of interconnection network that can potentially introduce variable latencies, the availability of the data at the orchestrator is non-deterministic. Hence, we need capability to wait for the data to be available and once an operation receives all its inputs, the operation can be scheduled for execution. Thus, irrespective of the execution paradigm, we need a variant of waiting and matching unit that can accumulate all the operands required by the macro operation and schedules it accordingly. Even in a control flow based execution model, in the presence of non-deterministic latencies, it is required to check if the contents in the register are valid before launching an operation. The only way to avoid such a kind of waiting for the data is to relegate the responsibility (of ensuring that the data is available for consumption) to the previous operation. Stated in another way, the completion of the current operation should also ensure that the data is available for the next configuration. It is similar to the concept of overlays in operating systems. At the end of current operation, the instructions are overlayed by a new set of instructions. In this model of execution, it is difficult to exploit dynamic parallelism available in applications. Further, this can be modeled as a special case of dataflow execution paradigm and hence the compiler can be targeted to such a model by suitably choosing the mapping modules. The requirement of being able to wait for the data and ability to exploit parallelism makes dataflow execution a natural fit. Semantics of the dataflow execution model provide an elegant way to exploit various kinds of parallelism available in the application.

In a dataflow execution model, each operation has its associated context memory, to which the data tokens are written. In strict-dataflow execution semantics, an operation is ready to execute after it receives all data tokens. In non-strict dataflow execution model, the operation can execute partially as and when inputs are available. We do not follow such a model, since it can potentially lead to underutilization of resources as resources are allocated per macro operation. The context memory management can be handled by hardware or can be controlled by the compiler. Depending on who manages context memory there are two variants of dynamic dataflow execution paradigm. Hardware managed context memory scheme is similar
2.4 Compiler Design Considerations

The compiler targets an architecture that follows a data driven synchronization model on the reconfigurable fabric and dynamic dataflow execution paradigm at the orchestrator. The abstract model of hardware is shown in figure 2.4. This model acts as an interface between the compiler and the micro architecture description. The abstract model we presented is generic enough to cover many instantiations of a CGRA to enable customization and design space exploration. Having defined the target of the compiler we now discuss the challenges faced when compiling onto such a platform.

Synchronization Mechanism:
- Data Driven
- No Flow Control

Dynamic Dataflow Execution Model

The compiler is responsible for transforming an application specification in imperative language onto CGRA, employing dataflow execution paradigm. There are two aspects for such a compilation trajectory viz. converting an imperative semantics to a dataflow graph, partitioning the dataflow graph, and enabling communication across these partitions. We delve into these aspects in this section.

2.4.1 Imperative language to Dataflow execution

Transforming an imperative language specification to a dataflow execution environment has been studied earlier (Beck et al., 1991) in the context of general purpose processing. Though at a high level the techniques employed are the same, they need to be fine-tuned for the CGRA. In this section, we...
Design of the Compiler

look at these aspects in detail and motivate various choices that we have made for transforming a C specification into a dataflow graph.

- **Ordering of Operations:** Imperative languages have a notion of order that is absent in the dataflow graph. For operations that do not have any side effects or modify a global state, data dependences enforce a partial order that is sufficient for correct execution. Operations like load-stores, function calls, and other console Input/Output (I/O) operations modify a global state and data dependencies alone are insufficient to ensure correct execution order. For example, consider a code sequence shown in figure 2.5(a), where the store instruction precedes a load instruction in the program order. If addresses for load and store operations **may** point to the same location, order needs to be maintained for ensuring correct execution. Figure 2.5 shows a dataflow graph where only the data dependencies are added. It is evident that data dependencies alone are insufficient and load could potentially overtake a store in such a dataflow graph. Hence, for such operations **precedence edges** are added to enforce order to ensure correct execution. It is to be noted that the in order issue of loads and stores does not automatically imply in order service by the memory subsystem. In the context of CGRA, the time between the firing of the operation and the service of the operation depends on the time spent to route the request to the appropriate memory unit. Existence of a precedence edge implies that the dependent instruction should be serviced by the memory subsystem only after the preceding instruction has been serviced. In dataflow execution models, memory operations are typically handled as split-phase transactions [Arvind et al. (1987)]. The LSU sends a response once the request has been serviced. In the absence of such a support, special schemes need to be devised depending on the capabilities of the architecture. We describe one such scheme in section 4.2.1.

Another approach for ensuring memory order is to assign unique identifiers to each load-store instruction such that these identifiers reflect the program order. [Smith et al. (2006a)](smith2006a) employ such a scheme to ensure correct ordering. Hardware is expected to process all these memory requests in the specified order. If equal numbers of memory operations are not available on all paths leading to a merge point (i.e., then and else clause of if statement) additional instructions are introduced such that memory operations receive a unique load-store identifier irrespective of the path taken. Such a scheme requires a centralized controller that processes all load-store requests and is complex to implement it efficiently in a distributed system. Further, it is not possible to use alias information to sequence different sets of load-store instructions. [Swanson et al. (2007)](swanson2007) use the notion of predecessor and successor instructions to address this limitation. However, it requires
2.4 Compiler Design Considerations

b = a + 20
c = a * d + b
store c, addr
e = load addr1

Figure 2.5: An example of transforming a code fragment into a dataflow graph (a) shows the code fragment that is being transformed to dataflow graph (b) Dataflow graph with only data dependencies (c) dataflow graph after adding precedence edges

additional support in hardware to match the predecessor instructions. This method of tagging a load-store instruction has potential to improve performance by eliminating the round trip latencies incurred when precedence edges are employed. However, precedence edges are simple to implement in the hardware. We use a modified scheme of adding precedence edges that eliminates the round trip latencies incurred. This scheme is described in section 4.2.1.

• Conditional operator: In traditional dataflow graphs, conditional operator is implemented using switch nodes (Arvind and Nikhil, 1990) that conditionally steers the data on one path based on the predicate value. In such graphs, data is delivered only when an operation is required to be executed. A naive placement of switch nodes can potentially limit the parallelism and add significant overhead. Beck et al. (1991) present an algorithm to eliminate redundant switch instructions. Even in such a dataflow graph, multiple switch nodes need to be employed when a data is used in a nested hierarchy of conditional statements as shown in figure 2.6. If-conversion proposed by Mahlke et al. (1992) is another way of converting control flow into dataflow. In this scheme, instead of predicaing the delivery of the data, instructions are predicated. The predicate is computed by compounding all appropriate predicates. In such an implementation, the computation of predicate can be significant overhead. Kathail et al. (2000) suggests several optimizations like wired operators, complementary predicates to address this problem.
\[ b = 20 \]
\[
\text{if } ( p == 1 ) \\
\quad a = 10 \\
\text{if } ( p1 == 1 ) \\
\quad a = a + 10 \\
\quad b = b + 20 \\
\text{else} \\
\quad a = a \times 30
\]

Figure 2.6: An example showing conditional operator implemented using switch nodes (a)Code Fragment (b) Dataflow graph implemented using switch nodes

[Smith et al. (2006b)](2006b) use nested predicates to reduce the overhead of predicate computation. An example of the same is shown in figure 2.7. The example does not show the merge nodes for the outer-level of nesting. In this method, instead of computing a compound predicate directly, the conditional operation itself is predicated by an outer level of nesting. This eliminates the need for compounding the predicates explicitly. Further, same instruction can be predicated from multiple predecessors provided only one of them delivers the correct predicate. This eliminates the need for an explicit OR instruction to compound the predicates. When instructions are predicated instead of data delivery, data could be delivered even when the instruction is not required to execute i.e. it is on a not-taken path. In such cases, to maintain a consistent state, these data have to be cleared before releasing the resources. When nested predicates are employed, it is more complicated to device a mechanism to clear data, as the arrival of data depends on the execution path. Thus, employing nested predicates comes with its own overhead of clearing the unnecessary data needed to maintain a consistent state. However, it has the advantage of reducing the number of switch nodes required when the data has long live ranges and is used in a deeply nested code. We employ a hybrid scheme to reap the benefits of both. We deliver data only when required by employing predicated steer nodes for intra-partition communication and we employ
2.4 Compiler Design Considerations

if ( p == 1 ) c = a + 10 if ( p1 == 1 ) d = b * 20

Figure 2.7: Example of nested predicates (a) Code Fragment (b) shows adding predicates by compounding the predicates (c) Adding nested predicates

nested predicate scheme for inter-partition communication. This has the advantage of keeping the fabric simple, since it does not require capability to clear the data. It also reduces the number of switch nodes for variables that are live for long intervals.

- Loops: Unlike simple operators and conditional operators discussed so far, loops execute the same set of operations multiple times. Hence, loops require special handling in the context of dataflow execution paradigm. Consider for example, a dataflow graph shown in figure 2.8, it is clear that the index increment can execute much faster than the computation itself. Hence, in the absence of any additional control, it will lead to inconsistent execution. This section discusses various aspects related to execution of loop.

Figure 2.8: Example of a loop implemented using switch nodes
Dynamic Instances of Loop: From the above example, it can be clearly seen that it is necessary to uniquely identify different instances of the loop that are active simultaneously. We first present different schemes employed in a dataflow execution paradigm and then discuss the suitability of each of these in the context of CGRAs.

- In tagged-token dataflow execution, each data is tagged appropriately to distinguish between various iterations. An operation is ready to execute when operands with identical tags are available on all input arcs. Such a scheme is known to be hardware intensive and is not practical to implement such a scheme on fabric. However, the tagged-token based scheme can be implemented at the orchestrator. Since orchestrator works at the granularity of macro operations, the overhead of tag generation and matching is amortized over multiple operations.

- In explicit token store (ETS) dataflow architectures, the responsibility of handling multiple instances of loops lies with the compiler. In this scheme, compiler requests for a context memory upon entry to the loop and all the data required for executing the loop is copied into the context memory. Once the context memory is allocated, the entire loop runs in that context. Depending on the parallelism available, compiler can request multiple context memories to be allocated, thus exploiting loop level parallelism.

- A static dataflow based scheme can be used to manage loops on fabric. In this scheme, a single instance of the iteration is allowed to be active at a time. This is implemented using acknowledgement signal from the consumer to the producer. A producer is ready to produce a data only when all consumers are ready to consume the data.

It is evident that ETS and TTDA schemes require hardware support and is not practical to implement these schemes on the fabric. They are well suited to be employed at the orchestrator. Though static dataflow based scheme is not complex to implement on fabric it requires as many acknowledgement tokens as data tokens. Hence, in section 6.1, we propose schemes that are customized for the CGRA. One of these schemes can be chosen depending on the application characteristics and performance requirements.

Loop Throttling: In ETS based execution paradigm, compiler controls the number of loops that can be executed in parallel. Depending on the application and the resources available in the hardware, compiler can choose an appropriate number of iterations that can run in parallel. However, in tagged-token based execution model, loops are instantiated dynamically. Culler (1985) shows that in the absence of a throttling
 scheme, an uncontrolled instantiation of loops can lead to severe pressure on the resources and eventually cause a deadlock. Further, the number of bits available in the tag also limits the number of iterations that can run in parallel. Compiler should ensure that no more than \( k \)' iterations could go in parallel by employing a scheme similar to that of k-loop bounding proposed by Culler (1985).

**Loop invariants:** A loop might have several data tokens that are invariant throughout the execution of the loop. This data can be either delivered repeatedly or can be stored and reused. In the tagged-token dataflow architecture, this data is referred to as loop constants and are re-circulated with a different tag for a new iteration. Though we employ tagged-token based execution model at the orchestrator, in the context of CGRA, it is more efficient to store these data tokens and deliver them when required. When the loop exits, this storage needs to be released. We discuss the mechanism to release these resources in section 5.2.1.

- **Function Calls:** Supporting function calls on fabric would require ability to suspend the existing configuration, execute the function, and restore the earlier configuration. Since having such a support on fabric is expensive, we do not support function calls on fabric. Function calls are supported only by the orchestrator. The allocation of the new context memory can be handled by the hardware by assigning a new function invocation identifier from the pool of unused identifiers or by the compiler by requesting a new set of context locations that will be used by the function call.

- **Other Constructs**
  - **Array References and pointers:** Array references and pointers are handled through memory references. Array references that can be transformed into scalar variables are converted to scalar variables to avoid load-stores and instead communicate on fabric. Alias information is used to expose more parallelism by removing precedence edges between memory references that do not alias. Memory operations that are known to point to different locations can execute in parallel.
  - **Function pointers:** Though we do not currently support function pointers, it is possible to handle them with the support of orchestrator. A function pointer refers to a function identifier in the context of CGRA, and the orchestrator is responsible for retrieving the contents of the appropriate function pointed by the function ID. When a function pointer is assigned, compiler is responsible for populating it with correct function ID.
2.4.2 Dataflow Execution on a CGRA

The important components of dynamic dataflow execution paradigm are the waiting and matching unit and the tag generation unit. Macro dataflow execution presents a different set of choices and new challenges in implementing them. We discuss these in the following sections. Macro dataflow execution (Sarkar and Hennessy, 1986) and dynamic dataflow execution (Swanson et al., 2007) have been studied in the context of General Purpose Processors (GPPs). When compared to GPPs, CGRAs employ simpler hardware and hence relegating more responsibilities (like resource management, instruction loading) to the compiler. The complexity of the compilation process is also due to the distributed nature of the reconfigurable fabric, which can introduce variable latencies.

- **Operation completion:** In macro dataflow execution, each macro operation comprises of multiple operations. It is necessary to detect the completion of a macro operation, since the resources allocated to it need to be reused. Completion of a macro operation can be defined based on the number of outputs produced or as the completion of execution of all operations in a macro operation. Further to ensure correctness, it needs to be ensured that there is no data in transit, when the operation is declared complete, i.e. the data produced by a macro operation (that is not meant for orchestrator) needs to be consumed by the same macro operation before resources are released. If this is not ensured, there is a possibility that the data appears after a new macro operation is loaded. The former definition of completion of operation requires hardware support to keep track of number of outputs generated to declare that the operation is complete. Since, the philosophy of CGRA is to keep hardware simple, we do not assume any hardware support for this purpose. Hence, we use the latter definition of completion. Hence, it is the responsibility of the compiler either to execute or purge every operation within a macro operation. This in combination with employing nested predicates requires the compiler to do special processing to purge operations on not-taken paths. For example, in figure 2.7, the execution of the multiplication is controlled by two predicates $p$ and $p1$. When $p$ evaluates to false, additional instructions are required to purge the multiplication operation.

- **Purging:** As mentioned earlier, we deliver loop invariants once and use them multiple times. Once the loop has finished execution, the loop constants need to be cleared. Compiler is responsible for generating a purge request and orchestrator should be capable of servicing it.

- **Terminating Macro operations:** As discussed earlier, we deliver data directly instead of employing switch operation to reduce the overhead.
of number of switch operations added. Hence, it is necessary to clear the data if a macro operation is not required to be executed. Compiler issues a purge request along with a count that specifies the expected number of inputs for a macro operation. Since we use a nested predication scheme, the macro operation might not receive all the inputs if it is on a not-taken path. Hardware should have support to wait for the expected number of inputs and clear the data. It can be seen that this mechanism is similar to the scheme used for selecting a macro operation for execution. Instead of launching the macro operation, the data is cleared and no further action needs to be taken. We term these requests as termination requests to differentiate them from the purge requests for loop constants. We describe the compiler support for purging and termination in section 5.

- **Tag generation:** In tagged token dataflow architectures, tags consist of four parts viz., invocation identifier, iteration identifier, code block and the instruction identifier (Arvind and David Culler, 1986). In the context of macro operations, the static part of the tag contains the macro operation identifier and the operand number of the macro operation. In the tagged token architecture, a code block is defined as a region that is acyclic or has a single loop. Thus, the iteration identifier specifies the dynamic instance number of the code block containing a loop. When there are multiple levels of nesting, an invocation identifier is required in addition to the iteration identifier. A central manager generates unique invocation identifier. Invocation identifier is reclaimed when the code-block finishes execution. In the context of CGRA, to reduce the pressure on the central manager, we use instance identifiers for four levels of nesting. In section 5.3.1, we show a simple scheme to manipulate these iteration identifiers, using shift and increment operators to obtain a new iteration identifier.

- **Reuse of context memory:** In compiler directed use of context memory, enabling the reuse of context memory within a loop/function is the responsibility of the compiler. Compiler can reuse the context memory allocated to a macro operation after the macro operation has completed its execution. However, when a macro operation is purged, compiler cannot determine when the hardware has actually processed the termination request because of the variable latencies that cannot be determined at compile time. Variable latencies can be incurred due to the presence of NoC, load-store delays etc. Consider for example, a scenario shown in figure 2.9. For the sake of simplicity, let us assume that each basicblock shown in the figure is a macro operation. As mentioned earlier, each macro operation is associated with a context memory location. Hence, in this example, each basicblock should be
assigned a context memory location. Let us consider the reuse of the context memory allocated to $D$. Also, in this example let us say $A$ is producing data to $D$. This is indicated by a dotted line in the figure. If execution takes the path $A \rightarrow C \rightarrow D \rightarrow F$ then $D$ would have executed, and the context memory of $D$ is ready for reuse at $F$. Since $F$ is expecting data from $D$, $F$ is ready to execute only after $D$ has delivered the data to $F$. This implies that $D$ has been launched on the fabric and the context memory allocated to it is free. Now consider a scenario when execution follows the path $A \rightarrow C \rightarrow E \rightarrow F$. Since $D$ is not on the taken path, a termination request is issued for $D$ by $C$. $F$ is merge point and it expects data from one of the basicblocks $D$, $E$, or $B$. In this path since $E$ is executed, $E$ delivers data to $F$ and $F$ is ready to fire. The execution of $F$ is independent of the processing of the purge message of $D$. In other words, the execution of a partition is only dependent on the availability of the data and is independent of any terminating messages that have been issued. Hence, for any context memory that is allocated to a partition that receives a purge message, compiler cannot identify any point in the program beyond which it is safe to reuse the location. One possible solution is to estimate worst-case delays and choose a point based on these delays. However, in many cases it is not possible to have such an estimate and even when an estimate is available it would be highly pessimistic and leads to an inefficient implementation. To address this problem, we identify synchronization operations that wait for the earlier purge request to be serviced before they execute. Hence, compiler can reuse such locations beyond a synchronization point. In this case $F$ awaits a message from $D$ about successful termination. In section 5.2.3 we discuss this scheme in detail.
2.5 Summary

A high-level overview of various choices and challenges in transforming an application specified in an imperative language to a dataflow execution based CGRA has been presented. When transforming an application to a dataflow graph, precedence edges need to be added between operations that modify/read from a global state. To reduce the number of switch nodes, data can be delivered directly to the operation. This would require termination requests to be issued to clear the unnecessary data. Instead of circulating the loop invariants, they are stored and reused. This requires purge requests to clear the storage for loop constants. To keep the hardware simple, the responsibility of generating tags has been shifted to the compiler. Further, for applications that have deterministic behavior it is more efficient to perform resource management by the compiler instead of investing in hardware.
Chapter 3

Related Work

In this chapter, we present the details of some recent compilers built for CGRAs. The other relevant works, such as those relating to the transformation of imperative languages to dataflow specification, dataflow machines, configuration management etc. are presented inline, along with their respective descriptions. Most of the research in compiling for CGRAs focuses on accelerating application kernels that can be accommodated on the fabric. Though there have been attempts at accelerating larger applications, they are relatively few. Since the focus of this thesis is on partitioning and orchestration of these partitions, we discuss frameworks that support these features in detail. We discuss the strengths and deficiencies of these frameworks. We also present a brief overview of compilers that accelerate application kernels. Reconfigurable architectures are also used to synthesize custom functional units to accelerate frequently occurring code sequences [Hauck et al. (2004); Arnold (2005)]. We do not discuss such reconfigurable architectures in this chapter. Cardoso et al. (2010) presents a detailed survey about the compilers for reconfigurable architectures.

3.1 PACT-XPP

PACT-XPP (Baumgarten et al., 2003) comprises a reconfigurable fabric that contains 2-D array of functional units. The memory modules are connected to the periphery of this array. Each functional unit can be configured to perform a particular operation and continues performing the same operation until it is reconfigured the next time. The functional units are connected by a programmable interconnection network. Configuration manager is responsible for handling these configurations. The manager consists of a configuration cache to store the configuration data. Along with the reconfigurable fabric, the platform also contains an array of control processors.
that are custom made and used to accelerate sequential portions of the code. These control processors are referred to as FNC-PAEs. The interconnection employed on the reconfigurable fabric consists of two kinds of networks, one for exchanging data, and the other for exchanging events. Events are akin to the predicates employed for transforming control information to dataflow. The reconfigurable fabric implements static dataflow semantics, where the instruction readiness includes availability of data and the readiness of consumer instructions to receive the data. Thus, this platform implements a flow control mechanism and is capable of tolerating variable latencies and supporting arbitrarily complex loops. This is the strength of this platform.

The compiler for this platform accepts applications developed in C/C++ languages. Applications are profiled and code sequences are identified that are suitable to be mapped onto the reconfigurable fabric. The rest of the application that is not accelerated is compiled onto the FNC-PAE, the custom-made control processor available on the platform. Typically the innermost loops are the candidates to be accelerated on the reconfigurable fabric. XPP-Vectorizing Compiler (XPP-VC [Cardoso and Weinhardt, 2002]) is used to generate the executable for the reconfigurable fabric. It accepts a C specification. The C specification should explicitly indicate the input and output of the reconfigurable fabric using library functions provided. XPP-VC compiler transforms the C specification into a Control-Dataflow Graph (CDFG) and the CDFG is mapped onto the reconfigurable fabric. Special state machines are introduced for ensuring the correct sequence when accessing the memory. The mapping step assigns the arrays into RAMs available on the reconfigurable fabric. It then maps the CDFG onto the reconfigurable fabric. If the application cannot be accommodated on the fabric, the application is partitioned into temporal partitions such that each partition can be accommodated on the fabric. These partitions are single entry, multiple exit blocks. The communication between these partitions is provided through dual-buffers provided along with the reconfigurable fabric. For performance critical code sequences, the platform also provides an option to specify the application using Native Mapping language (NML). It also provides a library consisting of components that are used frequently.

The strengths of the framework include the ability to support multiple modes of reconfiguration, a complete software tool set to program the architecture, the ability to support arbitrarily complex loops. The disadvantage with this architecture is the lack of instruction buffers with each functional unit. This restricts the size of the application that can be mapped onto the reconfigurable fabric. For large kernels like kernels of cryptography domain, it requires frequent reconfigurations and leads to a sub-optimal implementation. Further, they do not have support for TLP.
3.2 RAW

RAW machines are an interconnection of homogeneous tiles. Each tile consists of a simple five-stage pipeline, along with register files, and on-chip memory. These tiles are interconnected using two different types of networks. One interconnection network supports static routing and the other supports dynamic routing. The static routing is enabled by providing programmable switches. The dynamic switch is a wormhole router that takes routing decisions based on the header of each message. These switches use blocking semantics to provide near-neighbor flow control. The switches block when they attempt to read from an empty input port or when they try to write to a full output port. This allows the tiles to work independently even in the presence of dynamic events and non-deterministic latencies. These tiles can access the register file belonging to other tiles through one of these networks. This introduces non-uniform latencies for register access.

RAW compiler can accept applications specified in FORTRAN/C. RAW compiler is implemented using SUIF compiler infrastructure. The compiler performs a space-time scheduling of instructions to exploit ILP. The compiler is responsible for scheduling the instructions within a basic block. It is also responsible for orchestrating the control between the basic blocks.

Scheduling of operations within a basic block is performed through several steps. Initial code transformations are performed by transforming the basic block into a static single assignment form. Then, instructions are introduced to read variables that are live on entry and to write variables that are live on exit. Since compiler is responsible for communication of data values across tiles, these instructions help to handle the communication in later steps of the compilation. After the initial code transformations, the instructions are partitioned to exploit ILP available in the application. Initially, the instructions are grouped together into clusters such that each cluster contains instructions that do not have parallelism or the parallelism available across those instructions is too fine-grained to exploit on the RAW tiles. This step assumes infinite number of tiles when grouping instructions into clusters. These clusters are merged to match the number of tiles available in the processor. Compiler is also responsible for partitioning the data across these tiles. The scalar values used within a basic blocks are handled similar to the transports in a dataflow model. The source tile explicitly sends data to the consumer tile. The values that are exchanged between basic blocks are handled by assigning a home tile for each variable. At the beginning of the basic block, the value of variables is transferred from its home tile to all the tiles that use the variable. The modified value of the variable is written back to the home tile at the end of basicblock execution. In the next step, the virtual data sets and the instruction streams are mapped onto the physical tiles. As mentioned earlier, the communication of data across the tiles is also facilitated by the compiler. Once the instruction streams and data sets are
assigned physical tiles, necessary code is generated to enable communication across the tiles. This is achieved by generating appropriate code for the static switches to route the data. In the last step, the instructions assigned for each tile/switch are scheduled using a variant of list scheduling.

To orchestrate the control between basicblocks, compiler uses a technique called asynchronous global branching. It is used to implement branching across all the tiles, using static network and local branching in each of the tiles. This allows the tiles to work independently on multiple instruction streams. The branch value is broadcasted to all the tiles using the static network. Then, each tile and switch individually performs a branch at the end of basicblock execution depending on the branch value. Correctness is ensured because of the static ordering property of the network. Due to this property, the order of events at each tile remains the same. Even in the presence of dynamic events, the delivery of the events can be delayed but will not affect the order of the events. Further, an optimization called control localization is proposed to reduce the overhead of asynchronous global branching. In this optimization, a code-sequence containing a branch is treated as a single unit for scheduling. To apply this technique, code-sequences that contain memory operations referring to a single tile are identified. Then these code-sequences are scheduled on that tile eliminating the need for broadcasting a branch condition.

The strength of this compiler is its ability to exploit ILP by employing space-time scheduling. The data sets are also assigned efficiently to improve the memory parallelism. Further, given the ability to process multiple instruction streams independently, TLP can be efficiently implemented. However, support to exploit pipeline parallelism is not available. Each tile employed on the RAW machine is a simple RISC core and is capable of fetching next set of instructions. Hence, RAW machine is capable of accommodating arbitrarily large programs unlike the traditional CGRA. Hence, the step of obtaining temporal partitions and orchestrating them is not required when compiling to the RAW architecture. In this thesis, we explore a different design point that employs simpler cores with a focus on exploiting fine-grained TLP and pipeline parallelism.

### 3.3 Montium

Montium tile consists of Communication and Configuration Unit (CCU) and a Tile Processor (TP). CCU is the interface with external components and it is responsible for fetching the configuration and communicating with other external components. The tile processor is the core execution engine and contains five ALU units. The memory bandwidth required for these units is provided by ten memory banks available on the tile processor. Along with these, it also contains a decoder and a sequencer to program these ALUs.
The configuration of the ALU is determined by one of the eight registers associated with each ALU. These registers are 37-bits wide and one of these can be chosen in every clock cycle. Decoder and sequencer together determine the choice of this register for each of the ALUs. Decoder stores patterns that determine the choice of the configuration register for each of the ALUs. It is capable of storing 32 patterns and sequencer chooses one of these patterns in every cycle. When the number of patterns or the number of configuration buffers is not sufficient for an application, partial or complete reconfiguration has to be performed. The configuration for other modules, like memory modules, interconnection network etc., is also provided in a manner similar to that of ALU configuration.

Montium Compiler maps the application described in C onto the Montium tile processor. As a first step, the compiler transforms the application into CDFG. This graph captures both the control and dataflow information. In the next step, the CDFG is partitioned into clusters, such that each cluster is capable of executing on an ALU. The main objective of clustering is to (i) minimize the number of ALUs required (ii) minimize the number of ALU configurations required (iii) minimization of the length of the critical path. Clustering step is implemented using a graph-covering algorithm proposed by Guo et al. (2003). The next step schedules these clusters onto the target architecture by considering the maximum number of ALUs available on the architecture. To do so, a level number is assigned to each cluster based on the input-output relation between these clusters. Clusters that have the same level number can execute in parallel. Each cluster requires a particular configuration data and the objective of this phase is to minimize the number of different configurations mapped to an ALU. Further, this step also tries to minimize the different patterns across the ALUs that belong to one tile processor. As mentioned earlier, the decoder is capable of storing a fixed number of patterns, and this step aims at minimizing these patterns. In the last step, variables and arrays are allocated to storage units. Once the variables are assigned, necessary code is generated by adding instructions to fetch the required data from the storage devices.

The strength of this framework is the architecture that can efficiently exploit ILP and provide good performance if applications can be mapped onto the Tile processor. However, the architecture has several constraints and it is difficult to build efficient algorithms to compile under these constraints. For large applications, it is difficult to reuse the patterns, and it may become an intractable problem to find clusters and schedule them efficiently. Further, it does not provide support to exploit loop-level parallelism or pipeline parallelism.
3.4 Ambric

Ambric consists of a set of independent RISC processors organized into clusters. Each cluster contains ALUs, register files and local memories. The clusters communicate by passing messages through special channels. These channels are responsible for synchronizing the clusters. The channels are point-to-point and capable of accepting one word at a time. These channels behave like FIFOs, and at the end of each channel there is a special register, which the processor can read and write like any other register. Each core can run at a different clock frequency. The synchronization between these different clock domains is achieved through chained-register channels (Butts 2007).

Ambric supports an object oriented programming model using Java (Butts et al. 2007). In this programming model, application developers express parallelism explicitly by instantiating multiple objects of a class. The programmer also specifies the communication that happens between these objects. An object can be executed on a single processor core or multiple processor cores depending on the complexity of the object. The framework provides a GUI to specify the objects and communications between them. It also accepts this specification textually using a language called aStruct. The specification of each class is similar to a traditional single-threaded code in Java. The communication channels automatically provide the necessary flow-control and synchronization required.

This architecture was proposed to accelerate massively parallel codes using parallelism at the level of threads. Hence, the focus is only on exploiting TLP. Further, the responsibility of mapping applications is relegated to the application developer.

3.5 ADRES

ADRES is an architecture that consists of a coarse-grained reconfigurable fabric tightly coupled with a Very Long Instruction Word (VLIW) processor. The architecture is designed such that the functional units are shared between the VLIW processor and the reconfigurable fabric. In the VLIW processor view, several FUs are connected through a single multi-ported register file. Some of these FUs can access load-store units depending on the memory hierarchy employed. In the reconfigurable matrix view, along with these functional units and register file, it also contains an array of reconfigurable cells, comprising FUs and register files. These functional units can be heterogeneous and support predicated execution. These reconfigurable cells are connected using a point-to-point interconnect. These cells also contain configuration buffers that specify the functionality of the FUs at every clock cycle. The execution of the VLIW mode and the reconfigurable mode are
mutually exclusive.

DRESC, the compiler of ADRES, accepts the application specification in C. IMPACT (Chang et al., 1991) compiler framework is used as a front-end that performs traditional optimizations and generates an intermediate representation called "lcode". This intermediate code is analyzed to identify application kernels that can be efficiently implemented on the reconfigurable matrix. The identified kernels are mapped onto the reconfigurable matrix using modulo scheduling techniques. The algorithm proposed by Mei et al. (2003b) solves the problem of scheduling, placement, and routing of the data simultaneously. To be able to place and route, modulo routing resource graphs are introduced to model the target architecture. These are a combination of modulo reservation tables employed in traditional modulo scheduling and routing resource graphs employed in placement and routing algorithms of FPGA. Simulated annealing techniques are used to arrive at an optimal solution. The rest of the application is compiled using traditional VLIW compiler. The focus of this compiler is to map the kernels efficiently onto the reconfigurable fabric. Kernels that cannot be accommodated on the reconfigurable fabric are not supported on this framework.

3.6 PPA

PPA architecture is similar to that of ADRES reconfigurable matrix. It has been enhanced with features to support pipeline parallelism available in multiple applications. Further, this architecture is also capable of adjusting to throughput demands at runtime by suitably adjusting the number of functional units/cores allocated to the application (Park et al., 2009). Similar to ADRES, they also employ modulo scheduling techniques to map application kernels onto the target architecture. They employ edge-centric modulo scheduling proposed by Park et al. (2008) to map the kernels onto the reconfigurable fabric. The idea behind this technique is that in an architecture like CGRA the routing resources are more important and hence the scheduling technique gives higher priority to edges as opposed to nodes in traditional modulo scheduling techniques. To tolerate the dynamic variance in the computation requirement the number of cores allocated to a particular task can be adjusted at run time. This is achieved with the help of virtualization controller implemented in each core. The ability to migrate the code to different number of cores requires the compiler to generate a schedule that can be dynamically mapped to different target arrays. This is achieved by generating a schedule for a single core, such that it can be expanded at run-time by vertically splitting the schedule to map onto multiple cores. Modulo scheduling technique is modified suitably to introduce additional constraints to enable the splitting of the schedule onto multiple cores. Similar to the DRESC, the focus of this compiler is also to accelerate kernels that can
only be accommodated on the fabric.

### 3.7 Black diamond

Black diamond is a compiler framework that is developed to target a family of CGRAs. The compiler accepts hardware description as input in the form of a graph called Graph with Configuration Information. These graphs capture the information about the hardware platform. The nodes of this graph correspond to different ports of the functional units and the edges represent the connection between these ports. There are two kinds of edges: virtual edges and physical edges. Virtual edges are the edges that are not programmable and are fixed between the ports. Physical edges are programmable edges that can be employed for routing the data. The configuration of these edges is specified as a part of the opcode of the functional unit to which they are connected. If the edges are connected to a multiplexer then the control bits determine the configuration of the edges. Further, the architectures typically have restrictions, like the number of ports in a register file. To capture these restrictions a Disabling Configuration Testing (DisCounT) and control edges are employed.

The compiler accepts the application specification in a C-like language and targets it to the architecture specified by the Graph with Configuration Information. The access to the memory and other external I/O has to be specified by the library calls provided. The application specification should also contain other architectural details like the registers employed for describing the loop counts etc. Loops are specified by employing branch statement and the register specifying the number of iterations of the loop. This specification is first translated into a dataflow graph and then it is mapped onto the Graph with configuration Information honoring the restrictions of the architecture. They employ shortest-path algorithm proposed by Hiroyuki et al. (2006).

### 3.8 Summary

In this section, we presented some of the recent work in the domain of compiling for coarse-grained architectures. PACT-XPP uses dataflow execution model for implementing reconfigurable fabric. The computation elements do not employ instruction buffer and may require frequent reconfigurations when accelerating large application kernels. The application is partitioned into multiple partitions when the entire application cannot be accommodated on the reconfigurable fabric. The orchestration mechanism employed to schedule these partitions lacks the support for exploiting parallelism across the partitions. We also discussed other compilers that map application kernels onto the reconfigurable architectures. These compilers focus on mapping
the application kernel onto the reconfigurable fabric and do not accelerate kernels that cannot be accommodated on the fabric. PPA and ADRES use modulo scheduling techniques to map application kernels. Black diamond requires the user to specify the details about mapping of operations onto the architecture.

Ambric is designed for general purpose domain with a focus on exploiting thread-level parallelism. The responsibility of identifying parallelism and mapping them efficiently onto the array of processors lies with the programmer. RAW targets general-purpose domain and focuses on exploiting \textit{ILP}. \textit{ILP} is exploited by employing space-time scheduling of instructions. RAW provides a compiler that accepts applications in C/FORTRAN and automatically maps it to the architecture. Both RAW and Ambric employ \textit{RISC} cores capable of managing their instruction streams unlike many \textit{CGRA}. In this thesis, we look at another design point that employs simple cores and the responsibility of managing instruction buffers is left to the compiler.

As mentioned in the earlier chapters, the motivation behind this work is to design a compiler that has ability to exploit fine-grained \textit{TLP} that exists in the applications. We want to support fine-grained \textit{TLP} that exists across function calls, loop-level parallelism and pipeline parallelism. Most compilers focus on exploiting loop-level parallelism that exists in the inner most loops in the nesting hierarchy. We want to extend this to outer levels of the nesting. To achieve this we assume dynamic dataflow execution paradigm to orchestrate the application partitions to enable efficient exploitation of fine-grained parallelism. In the rest of the thesis, we present a compiler with a focus on orchestrating applications and has ability to exploit different kinds of parallelism along with ability to use the domain specific units that may be present on the reconfigurable fabric.
Chapter 4

Compilation flow

In this chapter, we present the compilation process for CGRAs employing a dataflow execution paradigm. The orchestrator follows a dynamic dataflow execution paradigm. This allows us to exploit fine-grained TLP efficiently. The compiler is designed to target multiple variants of such CGRAs. Compiling onto such architecture differs from that of traditional compilation flow in two major aspects. The first difference is that application needs to be partitioned into multiple partitions such that, each partition can be executed atomically and the partitions are executed in a dataflow execution paradigm. Though some of the compilers targeting reconfigurable architectures perform this step, we need to add restrictions imposed due to dataflow execution making the task of partitioning further constrained. The second major difference is that of managing interface between these partitions. In most CGRAs, this interface is handled by the host-processor. In our case, the partitions are executed in a dynamic dataflow execution paradigm. Unlike the traditional dataflow execution paradigm, where switch nodes are employed to deliver the data to the destination operation we deliver the data directly. This reduces the number of switch nodes significantly. However, since the data is delivered directly, it is possible that the data is delivered for an operation that is not required to be executed. This introduces additional challenges in managing the resources at the orchestrator. This is described in detail in chapter 5. In this chapter, we present the high-level overview of the compilation flow and discuss the dataflow graph creation and partitioning the dataflow graph in detail.

4.1 Overview of compilation

A high-level overview of the compilation flow for our CGRAs (described in chapter 2) is shown in figure 4.1. LLVM (Chris Lattner and Vikram Adve)
infrastructure is used to transform the given application specification into SSA form. This phase also performs several traditional optimizations like loop unrolling, function inlining etc. The optimized SSA representation is the input for our compiler. The first phase of the compilation is to transform an application specification into a dataflow graph. Control information in the imperative specification is transformed into data dependencies by employing predicates. In the context of this work, the entire application needs to be transformed into dataflow graph unlike the case of HyperBlock formation where only a part of the application is transformed into dataflow graph. Thus, it is important to optimize the dataflow graph to reduce the non-compute nodes introduced in this process. In addition, hierarchical predicates are employed to reduce the complexity of computing predicates for each operation in the dataflow graph. The dataflow graph thus constructed is partitioned into multiple partitions. Each of these partitions is referred to as HyperOps. The HyperOps have to honor structural constraints imposed by the fabric and additional restrictions due to the dataflow execution paradigm. This makes the partitioning of the dataflow graph different from that of partitioning the application employed in other CGRA compilers. This step uses both the control information and the dataflow information. HyperOps formed in this step do not contain loops as no assumptions are made on the availability of a flow control mechanism on the fabric. However, in some scenarios, it is beneficial to execute loops on the fabric to achieve higher efficiency. The next step, merges HyperOps that belong to loops that are to be resident on the fabric and forms Loop-HyperOps. If the target architecture lacks support for flow-control, loops are transformed such that they do not require a flow control mechanism to execute correctly. Static dataflow architectures use acknowledgements between every pair of communicating vertices to execute loops without using a flow control mechanism in the hardware. However,
in such an implementation the number of acknowledgements is as high as the number of data tokens. Hence, such an approach is not employed in this work. Instead, three different techniques are proposed assuming different capabilities in the hardware. We discuss these transformations in section 6.1. In the next step, these HyperOps and Loop-HyperOps are mapped onto the reconfigurable fabric. Mapping of a HyperOp involves (a) Partitioning the HyperOp further, such that each partition can be executed on a single computation element (b) assign each partition to a computation element depending on the interconnection network available on the fabric. This step requires detailed knowledge about capabilities of the fabric. The partitions of HyperOp are referred to as partitioned-HyperOps or pHyperOps for short. We do not delve into the details of partitioning the HyperOp. Inter HyperOp communication is suitably handled depending on the orchestration mechanism employed by the target architecture. The main challenges when handling inter HyperOp communication is that of managing resources at the orchestrator. We discuss this step in detail in chapter 5.

4.2 Dataflow Graph

The first phase of the compilation is to transform application specified in C language into a DFG. A DFG is a directed graph that shows dependencies between operations. The vertices in the DFG correspond to variables in the application specification and edges represent dependencies between them. This notion of DFG has been derived from the well-behaved dataflow graphs defined by Arvind and Nikhil (1990). When we construct DFG from an imperative language specification, the dependencies could be data, control or related to ordering of the operation. Figure 4.2 illustrates an example showing various steps involved in constructing the DFG. As a first step, application specified in C is converted into SSA (Cytron et al., 1991) form using Low Level Virtual Machine (LLVM) infrastructure (Chris Lattner and Vikram Adve, 2004). In SSA form, each variable is defined exactly once and hence it is easy to identify data dependencies. It eliminates artificial dependencies such as anti and output dependencies. LLVM transforms the application into instructions that comprise regular arithmetic and logic operations, load-store instructions, branch instructions, and \( \phi \) operations. \( \phi \) operations are used to merge values at join points. LLVM also converts array references into scalar references whenever appropriate. Array references that cannot be converted into scalar variables are translated into load-store instructions. An operation that contains an array reference is converted into a load followed by the specified operation.

In the second step, dataflow graph is constructed by adding data dependencies. Since the application has been transformed into SSA form, it is easy to identify definition and use of each variable. This step creates a dataflow
\[ e = p \times q \]

- if (a == 10)
  - b = addr[m]
  - c = b \times d
  - addr[n] = c
- else
  - b = addr[0]
  - c = d \times b;
  - addr[n] = e
- e = c + f

An example code listing

\[ e1 = p \times q \]
- if (a == 10)
  - b = \text{ld} \text{addr}
  - c1 = b \times d
  - \text{store} \ c1, \text{addr1}
- else
  - b1 = \text{ld} \text{addr2}
  - c2 = d \times b1
  - \text{store} \ e1, \text{addr1}
  - c3 = \phi(c1, c2)
  - e2 = c3 + f

Figure 4.2: Various steps in converting an application specification in imperative language into a dataflow graph
4.2 Dataflow Graph

4.2.1 Memory Ordering

To maintain the correct execution order, precedence edges are added between operations affecting a global state. The notion of order within a basic block is obvious and we capture this information by adding precedence edges between appropriate operations. To expose memory parallelism, we do not maintain the program order (i.e., add precedence edges) between memory operations that do not alias. Since it is known that these operations point to different locations it is not required to maintain program order to ensure correct execution. Hence, precedence edges are added only between memory operations that may alias. Thus, at the end of this step we have multiple memory chains in each basic block. Each memory chain represents the program order between operations belonging to a particular alias set. These memory chains need to be linked with the memory chains (corresponding to the appropriate alias set) of the predecessor basic block to obtain a global order. If all predecessor basic blocks contain memory chains then it is trivial to link them. However, in the absence of memory chains in any of the predecessor basic blocks it is non-trivial to compute the correct set of memory chains that have to be linked to ensure correct execution. The problem of finding correct set of memory chains is identical to that of finding the set of definitions at a merge point when transforming to SSA form. Therefore, we map the problem of finding correct set of memory chains to that of SSA transformation. To achieve this, each memory chain is modeled by a read followed by a write to a variable. All memory operations in an alias set are treated as performing a read and write from the same variable. This is done so that dependencies are not added between memory operations that do not alias. The problem of finding the preceding operations translates to that of finding all reachable definitions for that variable. This is achieved through the process of SSA translation. An example is shown in figure 4.3. While our technique uses dominance frontiers to analyze memory dependencies, Beck et al. (1991) use a post dominator approach. Both techniques are equivalent, but the use of dominance frontiers allows us to reuse the SSA-based techniques. To add memory dependencies across functions, function call is treated as a memory barrier for all the alias sets that the function can potentially read/modify. This information can be obtained
Compilation flow

by inter-procedural analysis. Currently, compiler expects the user to specify this information. In the absence of such information compiler makes a conservative assumption that this function modifies all alias sets. Further, to reduce the number of memory dependency edges that are added we use ROBDD based techniques described later in this section.

Unlike other operations, load-store operations are not serviced by the computation element but are serviced by a LSU. Since the order of the memory operations needs to be maintained at the LSU, it is the responsibility of LSU to send a trigger to the successor memory operation. Thus, precedence edges are handled differently when compared to other data edges. When a memory operation executes, the memory request is routed to the LSU. In the second phase, LSU returns a response back to the subsequent memory operation that acts as a trigger to execute the next memory operation. The location of the subsequent memory operation is encoded along with the memory request sent to the LSU. In the context of CGRAs, LSU is typically present at the periphery of the fabric. This introduces a long latency between firing of two dependent memory operations. This latency can be reduced, if the network ensures in-order delivery from every computation element and the LSU. In the context of CGRAs, ensuring in-order delivery is a reasonable assumption, given the use of a simple network that typically does not support adaptive routing. In such a scenario, the latencies can be reduced by grouping dependent memory operations into a single computation element. Since in-order delivery is ensured from a computation element to LSU, the order of issue at the computation element and the order of service at the LSU will remain the same. Hence, a dependent memory operation can execute immediately after the preceding memory operation has executed. Even in such an implementation, precedence edges are required to ensure that the memory operations are issued in correct order. In this scheme, precedence edges are treated as normal data edges. Figure 4.4 shows an example of each of these scenarios. In the case 4.4(a), load-store ordering is maintained by adding precedence edges that are serviced by the LSU. In such a scenario, store can fire only after the load reaches the LSU and LSU returns a trigger to execute the store. It introduces seven cycles delay between the execution of load and store operation. Once the store executes, it takes another 4 cycles for the request to reach the LSU. In the case 4.4(b), load and store are assigned to the same computation element and store can execute one cycle after load is issued, since the trigger is delivered locally. Hence, store can be serviced in 5 cycles as opposed to 11 cycles in the earlier case.

When the number of memory operations in the same alias set is more than the number of operations that can be accommodated in a computation element, they need to be distributed across different computation elements. In such a scenario, additional control is required as in-order delivery can only be ensured between one pair of elements. In such a scenario, trigger from the LSU is required between the last instruction in the memory chain assigned
Figure 4.3: Example showing addition of memory dependencies (a) shows a BB with a memory chain. For finding the appropriate memory chains that need to be linked, every memory chain is replaced by instruction to read followed by instruction write as shown in (b). (c) shows a control flow graph after performing SSA translation. BB B3 has a \( \phi \) node, but does not have a memory operation. Instead of adding a dummy operation \( \phi \) nodes are iteratively replaced. (d) shows links added from all predecessors after iterative replacement.
Figure 4.4: Example showing the latency between dependent memory operations. In this figure each box represents a computation element. Box marked with LSU represents the load-store unit (a) shows the latencies when precedence edges are serviced at the LSU. The memory trigger is delivered from the LSU. Hence, there is a round-trip latency from producer to consumer. (b) Shows the latency of the operation when all memory operations in an alias set are assigned to the same computation element and hardware ensures in-order delivery between any pair of communication entities.

To one computation element and the first instruction in the memory chain assigned to the next computation element. This is implemented by adding a synchronizing load that executes once all the memory operations assigned to a computation element have finished executing. This synchronizing load returns a data token that acts as a trigger for the next memory operation that could not be accommodated in the current computation element. An example of the same is shown in figure 4.5. For the sake of simplicity, let us consider each computation element can handle three instructions. Since the memory chain contains four instructions, they cannot be placed in one computation element. Therefore, we need a synchronizing load between the memory operations assigned to different computation elements. The memory chain is modified to add a synchronizing load at third position as shown in figure 4.5. It is to be noted that when the memory chain consists of operations that access different memory banks, multiple synchronizing memory operations need to be added. In CGRAs, it is common to have multiple memory banks connected to different peripheral points on the reconfigurable fabric.

4.2.2 Predicates

In traditional dataflow computing, switch nodes are employed to capture control flow information. Switch nodes steer data to an operation only when
4.2 Dataflow Graph

**Figure 4.5:** Example showing the synchronizing memory operations (a) shows the memory operations with memory precedence edges added (b) shows the dataflow graph after adding the synchronizing memory operations. In both (a) and (b) only a part of the dataflow graph containing memory operations is shown.

required. Another mechanism of transforming control information is by employing predicates. In this case, data delivery is not predicated and instead the operations are predicated. In applications that are control intensive, the number of switch nodes forms a large percentage of the operations. For example, in inertial navigation system application switch nodes are as high as 45%. On the other hand, delivering data directly requires a mechanism to clear this data when the operation is not executed. The efficiency of this scheme depends on the mechanisms employed to clear the data. In scenarios where multiple data tokens can be cleared using a single instruction direct delivery may be beneficial. In the context of macro-dataflow execution model, a single instruction can be used to purge the data produced for one macro operation that corresponds to data for multiple individual operations. Typically, a macro operation receives a number of inputs and hence clearing the data at the level of macro operations is more efficient than clearing the data at the level of operation. Further, employing switch nodes to deliver data to macro operations would require the data to be steered through other macro operations. This increases the number of inputs to a macro operation, thereby reducing its size when number of inputs limits the macro operation. Hence, data is delivered directly to a macro operation and purge instructions are inserted at appropriate places to clear the data. Within a macro operation, switch nodes are employed to deliver data. DFG is constructed by employing predicates and data delivery is not predicated. Once macro operations are identified, data transfers within the macro operation are processed and
switch nodes are added appropriately. We employ a scheme described by Beck et al. (1991) to add switch nodes. In this section, we describe the scheme used to transform control flow to dataflow using predicates.

According to the imperative language semantics, an instruction is executed only when control reaches that instruction. Data for the instruction might be produced irrespective of whether control reaches the instruction or not. Therefore, while building a DFG, these control dependencies have to be marked explicitly by appropriately predicating the operation. For any operation $O$ in a basic block $B$, the predicate is boolean AND of all the basic blocks in the path from the entry basic block to the basic block $B$. However, the execution of the immediate parent of the basic block $B$ ensures that all the other predicates are True. Hence, we add predicates only from the immediate parent. In the case where there are more than one immediate parents (i.e. the basic block has more than one in-edges), the predicate added is a boolean OR of the predicates of all the immediate parents of the basic block $B$. Program semantics ensure that only one of predecessors will be active. Hence an explicit OR is not required for adding a predicate. If there are several predecessor basic blocks, the expression controlling the execution of operation $O$ involves several input predicates (therefore several predicate in-edges). Further, this scheme limits the parallelism as the basicblock waits until the parent basicblock has executed. To minimize the predicate in-edges and to improve parallelism ROBDD based optimization is performed. In order to minimize the expression for computing the predicate of $O$, the CFG of the application is transformed into a BDD. This BDD is reduced to Reduced Ordered Binary Decision Diagram (ROBDD) (S. B. Akers 1978). ROBDD gives the minimized expression for the predicate. Detailed explanation of this technique is presented in section 4.2.4. Figure 4.6 shows an example of such an optimization.

### 4.2.3 Steer Nodes

As described by Cytron et al. (1991), $\phi$ nodes serve as merge points. During execution, an appropriate value is chosen depending on the path taken. In the DFG, this information is captured by adding steer nodes in each possible path that deliver data to the merge point. We ensure that only one token is delivered by appropriately predating the steer node. Steer nodes are an overhead in terms of the number of nodes and in terms of the delay added. To reduce this overhead we use the same steer node for multiple paths if they share the same source. For example, consider a scenario shown in figure 4.7 at basicblock $B6$, depending on the path taken, variable $a$ gets the data of either $a1$ or $a2$. Further, the value of $a2$ is assigned to $a$ from two predecessor basicblocks $B4$, $B5$. In this example instead of adding two steer nodes (one in $B4$ and in $B5$ as shown in figure 4.7(b)) it is sufficient to add only one steer node from $B2$ as shown in figure 4.7(c). To identify the minimal set
Figure 4.6: Example showing predicate hoisting. (a) shows a control flow graph (b) shows a DFG after the control dependencies are added. n1, n2, n3 are nodes in basic blocks B4, B5, B3 respectively. (c) displays the DFG after predicate hoisting is performed. The ROBDD of the CFG will be node B1 hence only one control dependency is added.
Figure 4.7: Example showing Optimization on steer nodes (a) shows a control flow graph with partial instructions in basic block. (b) shows a DFG after the Steer nodes are added. n1, n2, n3 are nodes in basic blocks B4, B5, B3 respectively that deliver data to the steer node. (c) displays the DFG after ROBDD optimization is performed on steer nodes.
of the paths that require a steer node, we obtain ROBDD for the subgraph of CFG rooted at the immediate dominator of the basicblock. This scheme is explained in detail in section 4.2.4. Further, it is not always required to add a special steer node. An additional steer operation is not required either if there is only one consumer of the source node or the steer node is not predicated. Both these cases are illustrated in figure 4.8. In the example shown, \( a_3 \) receives values from \( a_2 \) and \( a_1 \). The only consumer of \( a_1 \) is \( a_3 \) and hence \( a_1 \) can be directly predicated instead of adding an additional steer node. In the case of \( a_2 \), predicate is not required and hence it can be directly delivered to the consumer.

### 4.2.4 ROBDD Optimization

In all of the earlier mentioned techniques of handling precedence edges, predicate edges and steer nodes, the common problem is to compute the minimal set of predecessors from which the edges are required to be added. The problem of computing the minimal set of predecessors can be mapped to that of ROBDD optimization used in the context of logic minimization. ROBDD is a well-known technique in the context of hardware synthesis. It is used for logic minimization and equivalence checking. We apply the technique in the context of CFG to obtain the minimal number of conditions that govern the execution of a basicblock. CFG can be viewed as a BDD by considering each basicblock as a vertex. However, in the context of CFG, the
order of basicblocks is important and needs to be maintained unlike the case of BDD. Hence when adapting the technique of ROBDD to our problem, the order of the variables (basicblocks) is pre-determined. Hence, the notion of ordering of the boolean variables does not arise in our case. In this section, we discuss ROBDD based technique to compute such a set. The general idea behind this technique is to build a BDD corresponding to the problem and obtain a ROBDD that gives the minimal set of the predecessors. The BDD is built by treating each basicblock of the CFG as a boolean variable. The edges are added depending on the edges between these basicblocks in the CFG. The annotation of the edges depends on the problem we are trying to address.

- When minimizing the number of predicates, a “true” edge is treated as an edge corresponding to 1 and a “false” edge as an edge corresponding to 0. An unconditional branch is treated as having two edges one on “true” and another on “false”. An Unconditional branch is equivalent to having both edges (1 and 0) pointing to the same node. The expression for execution of the basicblock is now given by the BDD. Thus, the ROBDD gives the minimal expression and thus determines the minimal set of nodes to be added.

- In the case of optimizing the number of steer nodes, ROBDD optimization needs to be performed for each variable that is the source of the $\phi$ operation. If a variable is assigned only from one predecessor, this optimization is not required. When a variable is assigned from multiple paths, this can potentially reduce the number of steer nodes required. In this case, the edge is annotated with a value of 1 if the variable under consideration is used on this path and 0 otherwise. The problem of adding precedence edges is also handled in a similar manner.
4.2 Dataflow Graph

Once a BDD is obtained, techniques described by S. B. Akers (1978) are applied to obtain ROBDD. Further, it is not required to consider the entire CFG for this optimization. The sub graph of the CFG rooted at the immediate dominator of the BB whose predicate is being minimized is sufficient. Since by definition (Cytron et al., 1991), all paths to any node in the CFG has the immediate dominator as one of the predecessors, it is sufficient to consider the subgraph rooted at the immediate dominator. An example of this technique is shown in figure 4.9. Figure 4.9(a) shows a CFG rooted at BB A, which is the immediate dominator of BB H. Figure 4.9(b) shows the BDD corresponding to this control flow graph. It can be seen that BBs B, D, E, F do not play any role in determining the execution of H. Hence, they are eliminated. Figure 4.9(c) shows the ROBDD corresponding to BB H. The ROBDD optimization when applied to all BBs can potentially reduce the number of predicates required, leading to a reduction in transports.

The plot in figure 4.10 shows the percentage reduction in the number of non-compute nodes in the dataflow graph due to the ROBDD optimization. The amount of reduction depends on the characteristics of the application. Deblocking-filter is a control-intensive application and hence we see a large reduction due to the ROBDD optimization. Most of the other applications have very less control and hence the scope of ROBDD optimization is limited. In AES-E application, we see a large reduction due to the large amounts of parallelism available. Since many operations are parallel, all these operations have to be predicated. Even when the number of predicates reduced itself may be less, the number of nodes that gets affected is large. Hence, we see a large reduction in non-compute nodes.

Figure 4.9: Example showing ROBDD optimization. (a) shows control flow graph rooted at A which is immediate dominator of H (b) shows the corresponding BDD and also the logic expression for execution of BB H. We use a to represent a true path from A and a to represent a false path from A. and (c) shows the ROBDD.
### 4.3 HyperOps

The idea of partitioning/delimiting regions in the application for exploiting fine-grained task/thread level parallelism has been used by several teams in various contexts. \cite{Sarkar1986} proposed the use of macro dataflow execution for multi-processors. In their work, application is partitioned into multiple macro operations with the aim of obtaining minimum execution time. Since their work is done in the context of multi-processors they do not have structural restrictions such as the number of operations that can be allowed within a macro operation or the number of inputs a macro operation can receive etc. The work uses a model to estimate the delays and the macro operations are grown as long as there is improvement in execution time. In our case, we cannot estimate the execution time precisely as the execution time depends on runtime variations. Further, in the work of \cite{Sarkar1986} macro operations are delimited entirely based on the costs of communication and computation. In our case, we also need to consider loops as a boundary of the partition. This constraint is due to the dynamic dataflow execution employed at the orchestrator. This restriction is same as the restriction placed on waves in Wavescalar \cite{Swanson2007} or code blocks in tagged-token dataflow architectures. In our case, we have additional structural constraints and the partitions should be convex as explained later in this section. Another very recent work by \cite{Zuckerman2011}, uses "codelets" in the context of exascale computing to exploit fine-grained task level parallelism. This work is targeted for multi-core processors that have much higher capabilities (in terms of execution of loops, memory...
4.3 HyperOps

4.3.1 Overview of HyperOp

A HyperOp is a region of dataflow graph that satisfies following properties:

- HyperOps are disjoint: Every operation in the dataflow graph is included into exactly one HyperOp. Including an instruction in more than one HyperOp poses challenges in delivering the data and in scheduling the appropriate HyperOp. In situations where it is efficient to include a basicblock in multiple HyperOps, code duplication is employed to replicate the basicblock appropriately. An example of the same is shown in figure 4.11. Figure 4.11 (a) shows the desired HyperOp partitioning where A-B-D is one HyperOp and C-D is another HyperOp. To handle such a situation D is replicated into D1 and D2 and HyperOps are
formed by grouping A-B-D1 and C-D2 as shown in figure 4.11(b).

Figure 4.11: (a) shows the required partitioning of the CFG where Basicblock D is included in two HyperOps. (b) shows the HyperOps formed after duplicating Basicblock D so that HyperOps do not overlap.

- HyperOp Interaction graph is acyclic: As described earlier (refer section 2.3.2) HyperOp launching follows strict function semantics and the execution of the HyperOp cannot be interrupted i.e. HyperOps are non-pre-emptible. These two features imply that the HyperOp Interaction graph should not contain any cycles to ensure schedulability. Please note that backedges delivering data to the next iteration are not considered while checking for cycles. This condition on HyperOp Interaction Graph implies that HyperOps must satisfy the convexity condition as discussed by (Sarkar and Hennessy, 1986). A HyperOp is convex if all the paths between any two vertices in a HyperOp are contained within the HyperOp.

- HyperOps do not contain loops: Since the fabric does not have support for tags or tag manipulation, the code-blocks that require tag manipulation instructions are the boundary of the HyperOp. Since loops require tags, we do not include loops in a HyperOp. In later chapters, we relax this constraint to form loop-HyperOps that can contain loops on the fabric. We discuss more about loop-HyperOps in section 6.1.

- Inputs for a HyperOp: Since a HyperOp follows strict function semantics, a HyperOp will be executed only when all the inputs are available. Consider a hypothetical scenario shown in figure 4.12. In the example shown the HyperOp H1 contains basicblocks from two mutually exclusive paths. Hence, the inputs are generated for only one of the two basicblocks. When the number of inputs expected by each of the basicblocks is different, additional inputs needs to be delivered to
maintain the same number of inputs irrespective of the execution path taken. In the example shown in figure 4.12 let us say, basicblock C receives three inputs and basicblock F receives four inputs. An additional input will be introduced on the path A→C to deliver four inputs even when this path is taken. Though these inputs will not be consumed, they need to be delivered for launching the HyperOp. The example shown is hypothetical and such a HyperOp is not advantageous since it is known apriori that only one basicblock would be executed. It is more efficient to group them into different HyperOps to reduce the launch overheads. Hence, we do not form such kind of HyperOps where basicblocks from mutually exclusive paths are included in the same HyperOp and the decision making basicblock is not included.

![Figure 4.12: A HyperOp that contains basicblocks from mutually exclusive paths](image)

- Structural constraints: HyperOp cannot use more resources than available on the fabric. Thus, the number of instructions that can be accommodated on the fabric limits the size of the HyperOp. Further, orchestrator needs to keep track of the inputs available for a HyperOp. Hence, the number of inputs allowed for a HyperOp is limited by the capacity of orchestrator.

Having listed down all the properties of a HyperOp, we present the formal definition of a HyperOp as follows:

**Definition:** A **HyperOp**, \( H(V', E') \) is a vertex induced subset of the DFG \( G(V, E) \). \( V, V' \) are vertex sets of DFG \( G \) and HyperOp \( H \) respectively and \( E \) and \( E' \) are edge sets of DFG \( G \) and HyperOp \( H \) respectively. The set of HyperOps is pair wise disjoint. Further, HyperOps should satisfy the convexity property. In this definition we use the notation of \( V_p \) for referring to a HyperOp vertex set and \( v_i \) to refer to a vertex in a DFG.

- \( V' \subseteq V \).
• \( \forall (v_i, v_j) \in E \) such that \( v_i, v_j \in V' \) \( \exists \) an edge \( (v_i, v_j) \) in \( E' \)

• \( \forall p, q \) where \( p \neq q, V'_p \cap V'_q = \emptyset \)

• \( \bigcup_{i=1}^{n} V'_p = V \)

• \( \forall v_i, v_j \in V'_p \) all paths from \( v_i \) to \( v_j \) should be contained in HyperOp \( H_p \)

### 4.3.2 An algorithm to form HyperOps

HyperOps can be constructed by grouping together either operations or basicblocks. Grouping of operations provides finer granularity for selection and hence gives more flexibility in forming HyperOps. However important information about mutually exclusive paths, control dependence etc., is not available directly in a DFG. When HyperOps are formed by grouping basicblocks this information can be used effectively to form efficient HyperOps. Further, in most of the cases operations within a basicblock have close dependences. Hence, it is beneficial to group all these operations into one HyperOp. In this thesis, we describe an algorithm to group together basicblocks for constructing HyperOps. This algorithm allows us to include a partial basic block when a HyperOp cannot include the complete basicblock due to structural constraints (i.e. the number of operations in a HyperOp crosses a pre-determined threshold). The ability to include partial basic blocks requires a mechanism to partition the dataflow graph. This mechanism is also essential when basicblocks are large and they cannot be accommodated on the reconfigurable fabric. In the implementation of current HyperOp formation algorithm, importance is given to ensuring correctness of HyperOps. Though some optimizations are considered (for ex: using a depth-biased heuristic to balance the parallelism and the number of inputs of a HyperOp) when forming HyperOps, there is still scope for further optimization. We could explore different method of partitioning in the case of basic block split, different orders of traversal depending on characteristics of the application etc. Another important factor that determines the efficiency of the HyperOps is the set of traditional compiler optimizations that were performed on the application. It is important to choose the correct set of optimizations to obtain efficient HyperOps.

The algorithm considers the basicblocks in a topological order and includes them into a HyperOp if the structural constraints are met. If a basicblock cannot be included into a HyperOp, a new HyperOp is created. To ensure that convexity condition is met, it is sufficient to consider basicblocks in topological order. Topological order ensures that all the predecessors of the basicblock have been considered and assigned a HyperOp. Hence, there can be edges only in one direction and thus avoiding any cycles in the HyperOp interaction graph. Basic blocks are considered in a depth biased topologically sorted order for inclusion into a HyperOp. When multiple
4.3 HyperOps

paths in the merge, topological order traverses both these paths before considering the successor basicblocks to the merge point. This increases the number of basicblocks that are complementary to each other (as they are included from different paths). We bias the search towards depth whenever possible (refer figure 4.13) to reduce the number of complementary basicblocks included in the HyperOp. Further, a basicblock is a candidate for inclusion into a HyperOp only if the HyperOp contains the immediate dominator of the basicblock. This check is performed to avoid mutually exclusive basicblocks being included into a HyperOp. It is inefficient to include mutually exclusive basicblocks, when it is known at compile time that only one of these basicblocks will be executed. Further, such a HyperOp will need additional processing to ensure all its inputs are available as discussed earlier. For example in figure 4.13(b) if the HyperOp marked in dotted lines is limited by structural constraints, then basicblock $H$ cannot be included into the HyperOp. A new HyperOp is formed and basicblock $H$ is included into the new HyperOp as shown in figure 4.13(c). The next basicblock in topological order is basicblock $C$. Though including basicblock $C$ does not violate convexity condition, we do not include it since basicblocks $C$ and $H$ are on mutually exclusive paths. Now let us consider a situation as shown in figure 4.13(d) where three HyperOps are formed, and basicblock $J$ is under consideration. We do not include it into HyperOp 2 or HyperOp 3 since immediate dominator of basicblock $J$ is not included in those HyperOps. In situations where one path of the branch is more likely to be taken, it may be efficient to include basicblock $J$ in the most frequently taken branch. However, as discussed earlier such a HyperOp requires additional processing for generating correct number of inputs and we do not consider such kind of HyperOps.

If a complete basic block cannot be added, a partial basicblock is included into the HyperOp such that it does not violate the resource constraints. This involves partitioning the dataflow graph of the basicblock. Partitioning the dataflow graph also needs to honor the structural and correctness constraints mentioned earlier. When partitioning the dataflow graph of a basicblock we consider operations in a topological order to ensure convexity constraint is met. Since we are partitioning the dataflow graph of a basicblock and the same predicate govern all operations in a basicblock additional checks to ensure all inputs are available is not necessary. To form valid HyperOps, it is sufficient to consider operations in a topological order and include them into a HyperOp honoring structural constraints. Unlike the earlier case (of grouping basicblocks), the number of candidate operations is large and a good heuristic is necessary to form efficient HyperOps. We employ a heuristic based on the number of parents included in a HyperOp to reduce the communication across HyperOps and to reduce the number of inputs for a HyperOp. The idea behind this heuristic is to add an additional input only when we cannot grow the HyperOp any bigger with the current set
Figure 4.13: (a) Example showing the order in which basic blocks are included. We go into depth first and include basicblocks $B$, $D$, $E$, $H$ before we include basicblock $C$ as indicated by the dashed line. (b) shows the candidate basic blocks for the partially formed HyperOp and HyperOp cannot grow because of structural constraints (c) shows the case where a HyperOp cannot be increased because of mutually exclusive paths (d) shows the case where a HyperOp cannot include more basicblocks because of the absence of immediate dominator.
of inputs. Since the number of inputs to a HyperOp is limited, we would like to maximize the number of nodes included into a HyperOp with each additional input. An example of partitioning the dataflow graph is shown in figure 4.14(a). It shows a dataflow graph and a HyperOp \( H1 \) and a partially formed HyperOp \( H2 \). There are four candidate operations to be included into HyperOp \( H2 \). Operations \( I \) and \( J \) do not have any parents in HyperOp \( H2 \). Among operations \( G \) and \( H \) Operation \( H \) is given higher preference as all its parents are from the partially formed HyperOp. This heuristic is naive and does not perform well in all situations. This heuristic tries to maximize the first HyperOp without considering the impact on subsequent HyperOps. In the set of applications we evaluate this algorithm did not prove to be a bottleneck as the dataflow graphs consisted of relatively small number of operations (around 600 operations). However, we observe a lot of scope for improvement for large dataflow graphs.

![Dataflow Graph](image)

Figure 4.14: Example showing the partitioning of dataflow graph

An overview of the algorithm that creates HyperOps is presented in algorithm 1. active_list maintains the candidate basicblocks list. It is updated whenever a basicblock is included into a HyperOp as shown in algorithm 2. To enforce the depth biased heuristic basicblocks are chosen from the end of this list. Since basicblocks are always added to the end of the list, considering the basicblocks in the reverse order makes the algorithm depth biased. Function choose_bb identifies a basicblock for inclusion into the HyperOp based on the correctness constraints. Structural constraints are evaluated in the main body of the loop. Function choose_bb is described in algorithm 3. Algorithm 4 describes the method to partition a basicblock. Here the operations of the basic block are considered in topological order and the dataflow graph is partitioned into appropriate number of HyperOps such that all HyperOps are within the specified resource limits.

### 4.3.3 Correctness of the proposed algorithm

In this section, we look at the correctness aspects of algorithm 1. We prove that the algorithm forms valid HyperOps that conform to the definition of
function create_HyperOp()
    /* start with entry basic block */
    active_list = entry_bb;
    while active_list not empty do
        /* Create a new HyperOp and perform necessary initialization */
        new_HyperOp = create a new hyperop
        while true do
            /* choose the basic block from active list that can be included into the current HyperOp based on correctness conditions */
            new_bb = choose_bb (active_list, new_HyperOp);
            if new_bb is NULL then
                break;
            end
            /* check for structural constraints after including this basicblock */
            if satisfy_resource_requirement(new_HyperOp, new_bb) then
                /* add the chosen basic block to the HyperOp */
                /* Update the active list by including newly enabled basicblocks into the list */
                update_active_list(new_bb, active_list);
            end
            else
                /* Partition the basic block into multiple partitions such that each partition is valid */
                partition_bb(new_bb, new_hyperop);
                break;
            end
        end
    end
end

Algorithm 1: An Algorithm to create HyperOp

the HyperOp.


Proof. The algorithm to form HyperOps groups together basicblocks to form HyperOps. It partitions the dataflow graph of a basicblock, if basicblock cannot be accommodated within a HyperOp due to structural constraints. We look at algorithms[1][4] to prove correctness.

- HyperOp is a subset of the dataflow graph. $V' \subseteq V$
This is trivially ensured as both the algorithms do not consider any operations that are not present in the DFG for inclusion into a HyperOp.

- HyperOp is a vertex induced subset of the DFG.
function partition_basicblock(basicblock, hyperop)
    /* add all operations which do not receive any inputs from
       within the basic block to the active list */
    active_list = populate_active_list();
    while active_list not empty do
        /* Choose an operation from the active_list based on the
           communication with the HyperOp */
        new_operation = choose_operation(active_list);
        if satisfy_resource_requirement(new_HyperOp, new_operation) then
            /* add the operation to the hyperop */
            new_hyperop = new_hyperop \cup new_operation /* Update
            the active list by adding newly enabled operations */
            for each child of the operation do
                if all parents of the child operation are processed then
                    /* Add child operation to the active list */
                    active_list = active_list \cup child
                end
            end
        else
            /* create a new HyperOp and add the operation to the
               new HyperOp */
            create_new_hyperop(new_operation)
        end
    end

Algorithm 4: An Algorithm to partition the basicblock into multiple
HyperOps

The algorithm only chooses vertices of a HyperOp. Edges are added into
the HyperOp after the vertex set for a HyperOp has been determined.
This property can be trivially ensured by adding all edges that exist in
[DFG] between each pair of vertices belonging to a HyperOp.

- The set of HyperOps is pairwise disjoint: \( \forall p, q \in V'_p \cap V'_q = \emptyset \)

The algorithm considers each basicblock exactly once. There is a check
in function update_list (algorithm 2 at line number 4) to prevent consid-
ering a basicblock multiple times. Depending on various constraints,
we choose to include this basicblock into an existing HyperOp or cre-
ate a new HyperOp. In either case, the basicblock is included into
exactly one HyperOp. Similarly, the algorithm that partitions DFG
(algorithm 4) also considers vertices exactly once and is included into
4.3 HyperOps

One HyperOp. Thus, the HyperOps formed are non-overlapping.

- Every vertex in DFG is assigned to one HyperOp. $\bigcup_1^n V'_p = V$.

The algorithm visits all the basicblocks while forming HyperOps. Every basicblock that is added to the active_list is considered for forming a HyperOp. We exit the function only when the active_list is empty (condition of while loop at line number 3 in algorithm 1). A basicblock is included into active_list when all its parents have been processed. Since the algorithm does not consider backedges, the graph that is being processed is acyclic. Hence, each of the parents will be eventually processed. The only other case when the basicblock is not processed is when the basicblock is not visited. This again is not possible since the algorithm starts from the entry basicblock and all basicblocks are reachable from the entry basicblock by the definition of CFG.

When partitioning the DFG of a basicblock also this is ensured. DFG of a basicblock is acyclic but is not necessarily weakly connected. The first step of the partitioning algorithm is responsible for adding the root of each of the weakly connected sub graphs into the active list and hence all operations are processed.

- HyperOps are Convex. $\forall i, j, v_i, v_j \in V'_p \forall$ all paths between $v_i, v_j$ are contained in HyperOp. In other words, $\forall i, j, v_i, v_i \in V'_p, \#v_k | (v_i \text{ is ancestor of } v_k) \land (v_k \text{ is ancestor of } v_j) \land (v_k \notin V'_i)$

We prove this by using proof by contradiction technique. For the purpose of argument, let us assume that there exists such a case i.e. There exists such vertices $v_i, v_j$ and $v_k$. Since $v_i$ is the ancestor of $v_k$ and $v_k$ is the ancestor of $v_j$, $v_k$ is the first vertex that is considered for inclusion into a HyperOp since we consider vertices in topological order. Let us say $v_i$ belongs to HyperOp $H_p$, $v_j$ cannot be processed before $v_k$ as $v_k$ is the ancestor of $v_j$. When $v_k$ is processed, there are two choices for $v_k$, it could be either included in HyperOp $H_p$ or assigned to a new HyperOp. The first case leads to a contradiction that $v_k$ does not belong to $H_p$. In the second case if $v_k$ is assigned to a new HyperOp, HyperOp $H_p$ containing $v_i$ is frozen and no more vertices are added to it. Hence when $v_j$ is processed it cannot be included into HyperOp $H_p$ which again leads to a contradiction. Thus, it follows that such a case is not possible and hence HyperOps are convex.

4.3.4 Complexity analysis of the algorithm

In algorithm 1, basicblocks are visited in a topological order and depending on correctness and structural constraints each basicblock is either included
Compilation flow

into the existing HyperOp or a new HyperOp is created. The complexity of topological sort on the controlflow graph is $O(V_{CFG} + E_{CFG})$, where $V_{CFG}$ and $E_{CFG}$ are the number of vertices and the number of edges in the control flow graph. Thus, the complexity of ordering basicblocks is $O(V_{CFG} + E_{CFG})$. Processing of a basicblock involves a check to see if both correctness and structural criteria are met before including it into a HyperOp. The correctness condition involves checking if the immediate dominator of the basicblock is a part of the HyperOp and if the basicblocks already included in the HyperOp belong to the same loop. Since we maintain hash tables to store this information, it is $O(1)$ lookup. When checking for structural constraints, we need to check for the total number of operations in the HyperOp and also for a limit on the number of inputs for a HyperOp. Checking for the total number of operations is also a constant order operation. The number of operations in each basicblock is maintained in a hash table. In addition, the number of operations in the HyperOp under consideration is also maintained. Thus, the complexity of the check for total number of operations is $O(1)$. However to check for the number of inputs, we need to traverse the dataflow graph of the basicblock and for each operation in the basicblock we need to check all the sources of the operation and check if the source is included in the HyperOp or not. If it is not included in the HyperOp, then this becomes an input for the HyperOp. In such a case, we need to further check if this input is already considered as an input due to some other operation. The inputs of the HyperOp are stored in a hashtable and it is $O(1)$ to check if the input is already added to the HyperOp. We can also see that such a check is performed for each edge in the dataflow graph, hence the total complexity would be $O(E_{DFG})$, where $E_{DFG}$ is the number of edges in the dataflow graph. Thus the overall complexity of the algorithm is $O(V_{CFG} + E_{CFG}) + O(E_{DFG})$.

### 4.3.5 Measuring the effectiveness of Algorithm

To measure the effectiveness of the algorithm we have to define a metric to measure how far we are from the optimal solution. The metric involves several factors like

- Size of the HyperOp and total number of HyperOps
- The amount of concurrency between them
- The amount of communication between the HyperOps
- Critical path of the HyperOp

We need to combine all these factors into a single metric to be able to compare two different set of HyperOps for the same application. The ideal metric would be the actual execution time required to execute these HyperOps.
However, the actual execution time may depend on several runtime characteristics and hence cannot be determined at compile time. However, we could construct an analytical model based on these factors, that tries to evaluate the execution time. The execution time of each HyperOp could be obtained by static timing analysis by assuming average latencies for operations that have non-deterministic latencies. The expected performance for a function is then estimated based on the concurrency between the various HyperOps. We may also use profile information to obtain the number of times each HyperOp would be executing. The expected execution time obtained by combining all these factors can be used to guide the partitioning algorithm to derive better partitions.

### 4.3.6 Target Specific Customizations

Once HyperOps are formed, additional processing is done to make it amenable for execution on the reconfigurable fabric. This processing includes:

- Processing intra-HyperOp data transfers and adding steer nodes to deliver data only when required.

- To detect HyperOp completion, it is required that all instructions in the HyperOp have either executed or purged. Compiler has the responsibility to purge operations on not-taken paths.

- Adding output propagate nodes depending on the maximum number of destinations permitted by the hardware.

We employ the technique described by Beck et al. (1991) to add steer nodes to deliver data only when required. Intuitively, we need a steer node if we are not sure that the consumer would execute whenever producer executes. At that point, we add a steer node and predicate it appropriately. Further, these nodes have to be added at multiple points in the paths leading from producer to the consumer. Beck et al. (1991) prove that it is sufficient to add steer nodes at all branch points in the iterated control dependent set of the current basicblock. Beck et al. (1991) also show that these are the minimal number of steer nodes required.

The next task is to add instructions to clear the operations on not-taken paths. The first step is to compute the set of operations that have to be purged at each branch point. The set of operations to be purged depends on the direction of the branch. Thus, the objective is to find the set of operations to be purged for each branch point and for each direction of the branch. To find such a set, given a pair of basicblocks $P$ and $P'$ and the direction of the branch $d$ we should be able to determine if $P'$ will not be executed after $P$ takes a branch in the direction $d$. First, we present an example and discuss various cases before describing it formally. Consider the example
Figure 4.15: Example control-flow graph to illustrate the computation of purge instructions

Consider the case of computing set of operations to be purged at basicblock $B$. Let us consider the case of determining if a purge instruction is required in basicblock $C$ to purge operations belonging to basicblock $B$. Basicblock $C$ is on a complementary path to basicblock $B$ and when basicblock $B$ executes a decision on the execution of basicblock $C$ has already been taken. Hence the operations of $C$ would have been purged already and are no longer required to be considered. In other words, we need to consider only those basicblocks that are reachable from $B$ when computing the set of operations to be purged. Now consider the case of basicblock $D$, which is reachable from basicblock $B$. The execution of the basicblock $D$ is determined by the branch decision of basicblock $B$. If the execution follows the path $B \rightarrow E$ then operations in $D$ have to be purged. Once $E$ executes, $D$ will not be executed. In the absence of loops, this check is sufficient for introduction of instructions to purge operations on not-taken paths. This is also sufficient, when the loops have a single exit point. In the case of loops that contain multiple exits (as shown in figure 4.15), the check for reachability alone is insufficient. To illustrate such a case, consider the case when execution takes the path $D \rightarrow B$. This edge is a backedge and it starts the new iteration of the loop. Since new loop iteration is starting, the operations belonging to the earlier loop iteration have to be cleared. Hence, when $D \rightarrow B$ path is taken we have to purge operations belonging to $F$. In other words,
if the branch taken is a backedge, then we have to purge all basicblocks belonging to the loop that are still reachable. If the basicblocks do not belong to the loop, they should not be purged as they might be executed once the loop exits. For example we should not purge operations of basicblock $G$ as this will eventually execute when the loop exits. To capture the scenarios we have described thus far, we introduce a term called direct-reachability as follows:

**Definition:** A basicblock $P$ is **directly-reachable** from basicblock $P'$ in the direction $d$ based on the following function:

$$
\text{direct-reachability}(P, P', d) = \begin{cases} 
\text{sameLoop}(P, P') & \text{if branch in direction } d \text{ is a backedge} \\
\neg \text{reachability}(P_{\text{succ}}, P') & \text{otherwise}
\end{cases}
$$

Where:

- $\text{sameLoop}(P, P')$ is true if both $P, P'$ belong to the same loop
- $P_{\text{succ}}$ is the successor basicblock in the direction of branch
- $\text{reachability}(P_{\text{succ}}, P')$ is true if $P'$ is reachable from $P$ in the CFG without considering backedges.

Having defined direct-reachability, a purge instruction is required at a basicblock $P$ in the direction $d$, for a basicblock $P'$

$$
\text{purge-basicblock}(P, P', d) = \text{reachability}(P, P') \land \text{direct-reachability}(P, P', d)
$$

The algorithm to compute the operations to be purged is shown in code listing 5. It takes two parameters, the basicblock and the direction of the branch. The reachability between each pair of basicblocks is computed prior to this algorithm. This code assumes the availability of the reachability information. When the reachability information is available, the complexity of this algorithm is of the order $O(V_{\text{CFG}}^2)$. $V_{\text{CFG}}$ represents the number of vertices in the CFG, i.e. the number of basicblocks in the application. The reachability itself can be computed using well-known algorithms like Floyd-Warshall and has a complexity of $O(V_{\text{CFG}}^3)$. The set of nodes to be purged depends on the direction of the branch taken. We compute two sets of operations for each conditional branch. The assumption here is that a conditional branch has only two targets. When we have multi-way branches, then the same algorithm can be extended to compute multiple sets, each corresponding to one direction of the branch. Once, we compute the set of operations that need to be purged, purge instructions are added appropriately. Instead of adding one instruction to purge one operation, we
purge all operations assigned to a computation element using one purge instruction.

```c
function compute_purgeset(cond_bb, dir)
    taken_child = child of basicblock in the direction dir;
    back_edge = if the direction of branch is the backedge of bb;
    compl_child = child on the opposite direction of dir;
    for each basic block (bb) reachable from the compl_child do
        if back_edge then
            if bb in the same loop as cond_bb then
                /* add all operations of the bb to the purge set */
            end
        end
        else
            if bb is not reachable from taken_child then
                /* add all operations of the bb to the purge set */
            end
        end
    end
end
```

**Algorithm 5**: An algorithm to compute the set of operations to be purged at each branch point

### 4.4 Summary

In this section, we presented the overview of compilation flow. The first pass of the compilation transforms the imperative specification into a dataflow graph. Then the application is partitioned into HyperOps such that each HyperOp can execute on the reconfigurable fabric. The dataflow graph of the HyperOp is further partitioned and mapped onto the CGRA. This chapter discussed dataflow graph generation and partitioning the application in detail. We then discussed target specific processing required for HyperOp. The important contributions of this chapter are 1. Applying ROBDD optimization in the context of CFG to obtain efficient dataflow graph 2. Deriving necessary and sufficient conditions for ensuring correct execution 3. Algorithm to obtain HyperOps using both controlflow and dataflow information.
This chapter presents the details of compiler support for HyperOp orchestration. The support provided by the compiler depends on the orchestration mechanism employed. We discuss two variants of dynamic dataflow based orchestration mechanisms in this thesis. The three major issues addressed in this thesis are

1. A mechanism to reclaim the resources at the orchestrator. Unlike traditional dataflow execution paradigms, we do not employ switch nodes; instead, we deliver data directly to the consumer. Hence, the problem of reclaiming the resources is unique to our platform. Compiler introduces "purge" requests to free the allocated resource by clearing the unused data. Since the delay between the generation of data and delivery of the data to the orchestrator is not deterministic, we cannot assume an order between the arrival of the data tokens and the purge requests sent. Hence, each purge request is annotated with the exact number of data tokens that would be delivered. We describe the mechanism to identify correct locations at which the purge requests have to be added and also the algorithm to compute the expected number of inputs at each of these points in section 5.2.1.

2. In compiler directed resource management, allocating resources at the orchestrator is the responsibility of the compiler. Hence, we need to evolve a mechanism to identify precise points at which the resources are free before reusing them. In the case of purge requests, due to the non-deterministic latencies it is not possible to determine when the resource is actually free. To overcome this problem we identify synchronizing points that wait for a feedback from the purged HyperOps. As we
describe later in section 5.2.2 not all points can act as synchronizing points. Compiler has to choose synchronizing points to ensure correct execution.

3. In the hardware controlled resource management, tags are employed to differentiate between data tokens that belong to different instances of the loop. This scheme is similar to tagged token dataflow execution ([Arvind et al., 1987]) proposed in mid 80’s. In the tagged-token dataflow architectures, the generation of tags is the responsibility of a centralized hardware unit. Other approaches like wavescalar ([Swanson et al., 2007]) based on tagged-token dataflow architectures also employ a similar scheme. In the context of CGRA’s such a central hardware entity will be expensive in terms of area and power consumed. In our implementation, we avoid the use of such a hardware unit by additional support from the compiler. Compiler provides hints to the hardware to generate tags efficiently. The details of this scheme are described in section 5.3.1.

In this chapter, we first present a brief overview of the orchestrator, summarize the responsibilities of the orchestrator, and then discuss the role of compiler in enabling the orchestration of HyperOps.

5.1 Overview of orchestrator

At a very high level, orchestrator is responsible for managing inter HyperOp communication and allocating required resources for HyperOps. When a HyperOp has to communicate with another HyperOp, the data is delivered to the orchestrator along with meta-data to determine the exact HyperOp to which the data has to be delivered. Using this meta-data, orchestrator updates the appropriate context memory location with the new data. The data required for each HyperOp is collected at a single location since it is easy to check for availability of data for a HyperOp. This memory location associated with each HyperOp is referred to as a context memory location. Each context memory location is capable of holding a specified number of data inputs and this limits the number of inputs for a HyperOp. Once required data is available for a HyperOp, the orchestrator is responsible for finding a suitable set of computation elements on the fabric and the HyperOp is launched onto the fabric. Compiler is expected to generate a resource requirement matrix that specifies the required set of computation elements and the interconnection between them. When executing loops, loop-invariant data is produced once and it has to be used for multiple iterations of the loop. There are two approaches for handling loop-invariant data (i)data can be delivered multiple times, once for each iteration of the loop (ii) it can be stored and retrieved whenever required akin to the Manchester machine’s
5.1 Overview of orchestrator

sticky token store. We employ the second scheme, as delivering the data multiple times can be expensive. The storing and retrieving of loop-invariant data is the responsibility of the orchestrator. Once the loop finishes execution, the loop-invariant data has to be purged. Similarly, when a HyperOp receives partial data but is on a not-taken path, the data corresponding to that HyperOp has to be cleared.

The context memory required for a HyperOp can be assigned at compile time or it can be allocated at runtime. For applications that have deterministic behavior and have fixed throughput requirements, compiler can determine the resource requirement. For such kind of applications, hardware complexity can be reduced by assigning the responsibility of resource management to the compiler. When applications have non-deterministic behavior and the parallelism needs to be unfolded at runtime, resources are best allocated at runtime. Hardware can control the resources at the orchestrator by providing resources for various instances of HyperOps when necessary at runtime. We refer to the former mechanism of orchestration as compiler directed resource management and the latter as hardware controlled resource management. Either of these orchestrators can be employed based on the use case. We describe the role of compiler in both these schemes in detail in this chapter. The responsibilities of the compiler are summarized below:

- Delivering data to another HyperOp: Depending on the scheme employed for orchestration, compiler has to generate necessary meta-data to identify the context memory location corresponding to the specific instance of the consumer HyperOp.

- Finding a location on the fabric: Compiler should generate a resource requirement matrix that specifies the number of compute units required for a HyperOp and the interconnection pattern required between these compute units. To generate such a matrix, compiler partitions the dataflow graph of the HyperOp into multiple partitions referred to as pHyperOps. Then the pHyperOps are mapped on to the fabric based on the interconnection network available and the communication pattern between pHyperOps. This step uses the dataflow graph of HyperOp as an input and does not affect the interface between HyperOps. Thus, this step remains the same in both the schemes. This step of the compilation is outside the scope of this thesis, [Krishnamoorthy et al. (2011)] provides further details of the partitioning and mapping of the HyperOps onto a fabric employing [ALU] interconnected using a [NoC].

- Terminate the HyperOps on not-taken paths: Compiler has to introduce instructions at appropriate locations in the application to purge HyperOps. The mechanism to purge the HyperOps remains the same in both the schemes. Depending on the orchestration mechanism employed, the delivery of the purge requests has to be handled appropriately.
• Handle Loop Invariant data: Loop-invariant data has to be identified by the compiler and such data needs to be tagged as loop-invariant data. Hardware takes necessary action to use this data for multiple iterations of the loop. Further, necessary instructions have to be introduced to clear the data upon loop exit.

In the rest of the chapter, we discuss in detail how the compiler handles each of these responsibilities. We first describe the various aspects that need to be considered when handling these tasks. Then, we present the compiler directed resource management followed by hardware controlled resource management.

5.2 Compiler directed resource management

In compiler-directed resource management, the compiler requests the reservation of a fixed set of context memory locations for use within a function. The compiler is responsible for allocation of context memory locations for HyperOps within the function. The context memory locations are reused by non-overlapping HyperOps. In such a scheme, data delivery to a HyperOp translates to that of writing data to a specified location. This eliminates the need for hardware required for determining the context memory location corresponding to the HyperOp. As we discuss later (in section 5.3), a lot of bookkeeping information needs to be maintained by the hardware, in order to determine the context memory location associated with a specific HyperOp instance. This renders the hardware complex. This scheme also reduces the latencies involved in exchanging data between HyperOps since the locations are identified at compile time. For most of the applications, that have deterministic characteristics this scheme is more suitable than the hardware controlled scheme. We discuss the details of the compiler role in enabling the orchestration of HyperOps using this scheme.

5.2.1 Terminating HyperOps on not-taken paths

As mentioned earlier (in section 4.2.2), data for a HyperOp is delivered directly without employing switch nodes. This is primarily done to (i) reduce the overhead of switch nodes added (ii) to reduce the number of inputs for a HyperOp, as employing switch nodes would require data to be routed through other HyperOps. When a data token has to be tunneled through a HyperOp, the data is not used in the HyperOp and it is an additional input to the HyperOp. It is our observation that in some cases the size of HyperOp is limited by number of inputs. Thus, this would further limit the size of the HyperOp. Hence, we do not employ switch nodes and deliver data directly to the HyperOp. Since the data delivery is not predicated, data might be available at the orchestrator even when the HyperOp is on a not-taken
path. Such data has to be cleared before the context memory location is reused either for a different HyperOp or for a different instance of the same HyperOp.

5.2.1.1 Computing Purge Points

Compiler adds instructions at appropriate locations to purge such HyperOps. To add purge instructions, we need to determine the HyperOps that are on not-taken path at each conditional branch. This information is not directly available from the information like dominators, post-dominators and dominance-frontiers computed in traditional compilers. In CFGs that have well-defined structure, the information about the basicblocks to be purged can be computed by subtracting post-dominator set from the dominator set. However, this does not work in all the cases since in some cases we need to purge basicblocks that are not dominated by the basicblock. Such cases are not rare and can be found in many of the optimized CFGs obtained after performing the simplification of the CFG. Further, all this information is available at the level of basicblocks and in our context we need to extend it to that of HyperOps. Hence, we need to formally define the purge points of the HyperOp. In this section, we define the purge points and present an algorithm to compute the same.

It can be seen that, the problem of computing the set of HyperOps to be purged at a branch point is similar to that of finding the set of operations to be purged at a branch point within a HyperOp. We borrow the terminology from section 4.3.6 to define the purge points for a HyperOp.

**Definition:** A HyperOp $H'$ has to be purged at HyperOp $H$ if branch in direction $d$ is taken

$$\text{purge-HyperOp}(H, H', d) = \text{reachability}(H, H') \land \text{direct-reaching}(H, H', d)$$

Where,

$$\text{direct-reaching}(H, H', d) = \begin{cases} \text{sameLoop}(H, H') & \text{if branch in direction} \\ \neg \text{reachability}(H_{succ}, H') & \text{otherwise} \end{cases}$$

Where, $H_{succ}$ is successor of HyperOp $H$ in direction $d$, $\text{sameLoop}(H, H')$ is true if $H, H'$ belongs to the same loop and $\text{reachability}(H, H')$ is true if $H$ is reachable from $H'$.

The definitions of reachability of a HyperOp and the direction of a branch of HyperOp are similar to that of basic blocks in a CFG. Since HyperOp is
a collection of basicblocks, HyperOps can have multi-way branches (For ex: H1 in figure 5.1). Hence, the set of purge instructions has to be computed for each direction of the branch. The target of the branch is computed using the control edges in the HyperOp interaction graph. HyperOp interaction graph represents the communication happening across HyperOps. The edges in the interaction graph can be control edges, data edges, or backedges. Control edges represent the control dependencies of the HyperOps and are derived from the control flow graph. Data edges are the actual data transfers happening between HyperOps. Backedges transfer data/control across iterations of the loop. The algorithm to compute the set of HyperOps to be purged is similar to that of the algorithm 5 that computes the set of operations to be purged.

5.2.1.2 Computing the number of inputs

Unlike the earlier case of purging within a HyperOp, the number of inputs received by the HyperOp need not be equal at all purge points. Further, because of non-deterministic latencies on reconfigurable fabric, data and purge requests can reach the orchestrator in any order. If the purge request reaches the orchestrator before all data tokens are available, then only a portion of the expected data is cleared leaving the context memory location inconsistent. Hence, we specify the number of inputs expected along with the purge request to a HyperOp. Compiler is responsible for determining the correct number of inputs generated for a HyperOp at each of these purge points. Further loop invariant data needs to be handled differently. When a HyperOp is a part of the loop, loop-invariant data should not be invalidated since it is required for next iteration of the loop. However, if a HyperOp belonging to a loop is purged because the loop is on the not-taken path, then loop invariant data has to be cleared. In other words, if a purge point is within the same loop as the HyperOp being purged then loop invariant data should not be cleared where as if the purge point is outside the loop, then loop invariant data has to be cleared.

Each of the earlier mentioned cases are illustrated through the example shown in figure 5.1, the solid lines show the data tokens and dotted lines show the control dependence between the HyperOps. Not all data edges are shown in the figure for the sake of simplicity. In this example, consider the purging of the HyperOp H6. It has to be purged at multiple points depending on the path taken during execution. It has to be purged at H1, H3 and H4 when execution follows the paths H1→H8, H1→H3→H7, or H1→H3→H4→H7 respectively. In the first case, no inputs are generated for H6 and hence a purge instruction need not be added. In the second case, when execution follows the path H1→H3→H5, one input is delivered from H2 to H6 and hence a purge instruction is added at H3. In the last case when execution follows the path H4→H7, two inputs will be delivered to HyperOp
### 5.2 Compiler directed resource management

The number of inputs available for HyperOp $H_6$, at each of these purge points is different. Delivery of purge messages at the orchestrator follows all the data tokens for a HyperOp. To ensure correct execution, compiler has to determine the correct number of inputs expected for a HyperOp at each of its purge points and annotate the purge instruction accordingly.

Further, compiler needs to account for the loop invariant inputs separately. The number of loop-invariant inputs is required when terminating a HyperOp from within a loop. For example in figure 5.1, consider the loop consisting of HyperOps $H_8$, $H_9$, $H_{10}$. In every iteration either $H_9$ or $H_{10}$ will execute. Hence, one of these has to be purged at the end of each iteration. However, loop-invariant data should not be cleared, as it is required for subsequent iterations. In such cases, the expected count that is delivered along with the purge messages should not include the count for loop-invariant data.

![Figure 5.1: Example showing a CFG and HyperOp Interaction Graph](image)

We compute the aggregate number of inputs that should have been generated for each HyperOp at each purge point. To do so, first we record the communication that happens between every pair of HyperOps. This records the information about the inputs expected for a HyperOp and the source HyperOp from which the input is produced. Inputs to a $\phi$ operation need not be considered since $\phi$ operations are employed at merge points and data delivery is predicated. Inputs for a non-$\phi$ operation can only be generated by a basicblock that dominates the basicblock containing the operation. In other words, inputs that have to be purged can be generated only by basicblocks in the iterated dominator set of the current basicblock. Thus, to obtain the aggregate number of inputs at purge points it is sufficient to look at the inputs generated by the iterated dominator set of the HyperOp. The dominator set of a HyperOp is a set of HyperOps corresponding to the union of Iterated dominator sets of the basicblocks contained in the HyperOp. When basicblocks are partitioned into multiple HyperOps, then we need to
This function takes three arguments: *
Inputs:
/* num_ip contains the information about the number of inputs between every pair of HyperOps. It is indexed by source HyperOp Id and destination HyperOp Id to obtain the number of inputs to destination HyperOp from source HyperOp. */
/* bb_to_HyperOp_map has a mapping from the basicblock to the list of HyperOp(s) it belongs to. It is indexed using a basicblock and gives the set of HyperOps that this basicblock belongs to */
/* SrcHyperOpId is the source HyperOp from which we are computing the aggregate inputs */
/* DestHyperOpId is the Identifier of the HyperOp to which we are computing aggregate inputs */
Outputs:
/* aggregateInput contains the number of aggregate inputs between every pair of HyperOps. Similar to num_ip it is indexed by source and destination HyperOpId. */

function compute_aggr_ip(num_ip, cfg, bb_to_hyperop_map, src_hyperop_id, dest_hyperop_id)
  /* get representative basicblock of the HyperOp */
  dest_rep_bb = representative_basicblock(dest_hyperop_id);
  src_rep_bb = representative_basicblock(src_hyperop_id);
  /* get the Immediate dominator of the representative basicblock */
  Idom_bb = Idom(dest_rep_bb);
  while Idom_bb is not dest_rep_bb do
    /* get the set of HyperOp Ids corresponding to the Idom_bb basicblock */
    hyperop_id_set = bb_to_hyperop_map[Idom_bb];
    IteratedDomSet(hyperop_id) = IteratedDomSet(hyperop_id) ∪ hyperop_id_set
    Idom_bb = Idom(Idom_bb);
  end
  for each h in IteratedDomSet(hyperop_id) do
    aggregateInput += num_ip(h, hyperop_id)
  end
end

Algorithm 6: Pseudo code showing the computation of aggregate number of inputs received for a HyperOp, at each HyperOp
include all the HyperOps corresponding to that basicblock. More formally,

\[ \text{IdomSet}(H_i) = \bigcup_{j=1}^{n} \text{HyperOpIdset}(\text{IdomSet}(bb_j)) \]

where, \( \text{HyperOpIdSet}(bb) \) is set of HyperOps corresponding to the \( bb \)

Further, when HyperOps are formed using the algorithm described in section 4.3.2 there exists one basicblock in the HyperOp that dominates all other basicblocks belonging to this HyperOp. This basicblock is referred to as a representative basicblock of the HyperOp. The algorithm does not include basicblocks into a HyperOp if the immediate dominator of these basicblocks is not already present in this HyperOp. Hence, it is ensured that there is one basicblock that dominates all other basicblocks in the HyperOp. Pseudo code presented in algorithm 6 uses this information to compute the aggregate number of inputs between the specified HyperOps. The algorithm represents the core idea behind the computation of aggregate number of inputs. This algorithm computes the number of aggregate inputs between a given pair of HyperOps. However, since we need to obtain the aggregate number of inputs between multiple pairs of HyperOps, such an implementation would not be very efficient. In the actual implementation, we traverse the list of HyperOps and compute the aggregate number of inputs between all pairs of HyperOps in one pass. In our implementation, the number of inputs of the current HyperOp under consideration is updated by adding the number of inputs from its immediate dominator. Since, we process these HyperOps in a topological order, the immediate dominator would be already updated with the aggregate number of inputs from all its predecessors. Thus, it is possible to compute the aggregate number of inputs in one pass. To handle cases where a basicblock is split into multiple HyperOps, a pre-processing step is performed and the outputs generated by each of these HyperOps are aggregated into one particular HyperOp in this list. This does not disturb the correctness of the algorithm because by definition if a single instruction of a basicblock is executed then all the instructions in the basicblock will be executed. Hence, such an assumption would not lead to any inconsistencies. Further, it reduces the number of special cases that need to be considered while obtaining the aggregate number of inputs. The arguments for a function are treated as if they are generated by the entry basicblock thus avoiding special processing for the arguments of the function. The complexity of computing aggregate number of inputs is dependent on the data structures employed and the information available due to the earlier passes. The number of inputs between every pair of HyperOps is already available in our implementation. This information is updated when forming HyperOps. As mentioned earlier (in section 4.3.2), when forming HyperOps, we check for the total number of inputs for a HyperOp. In this process.
we determine the inputs of a HyperOp and the source HyperOp of these inputs. This information is recorded in a hashtable indexed by the source, destination HyperOp pair. Further, HyperOps are formed by traversing the CFG in topological order. Thus, the list of HyperOps is already available in topologically sorted order. Thus, the complexity of this algorithm involves in scanning through the list of HyperOps and updating the aggregate number of inputs. The complexity of this algorithm is linear in the number of HyperOps.

As mentioned earlier, we have to compute the aggregate number of inputs for normal and loop-invariant data separately. To achieve this, we record the number of loop-invariant inputs and the normal data inputs separately and aggregate them. Once we compute the purge points and the aggregate number of inputs for each HyperOp purge instructions are added. As mentioned earlier, a purge instruction is added for a HyperOp \( H \) at a purge point \( H' \) under following conditions:

- If HyperOp \( H \) and HyperOp \( H' \) belong to the same loop, only aggregate number of inputs is considered for adding a purge instruction. If the aggregate number of inputs is non-zero then a purge instruction is added with the appropriate count.

- If HyperOp \( H \) and HyperOp \( H' \) do not belong to the same loop, we consider both the aggregate number of inputs and aggregate number of loop-invariant inputs. If the sum of these two is non-zero then a purge instruction is added with this sum as the number of expected inputs.

### 5.2.2 Synchronizing HyperOps

So far, we have discussed the mechanisms of delivery of the data and clearing the data when it is on not-taken paths. The presence of variable latencies renders these events (of delivering the data and clearing the data) non-deterministic. For HyperOps that are on taken-paths this non-determinism does not pose a problem for reusing the context memory locations, since a HyperOp waits for all data tokens to be available prior to start of execution. The execution of the HyperOp implies that the context memory location assigned to the HyperOp is ready for reuse. On the other hand, non-deterministic latencies pose additional challenges when determining the reuse of a context memory location assigned to a HyperOp on not-taken paths. The context memory location assigned to such HyperOps can be reused only after the purge request has been serviced. In the absence of any additional information, it is not possible for the compiler to determine when the request is actually processed. Even when we assign a unique context memory location for each HyperOp, reuse happens across multiple instances of the same HyperOp. Such a reuse also can potentially lead to incorrect execution if the data is not cleared before the reuse.
To handle this problem, we designate some HyperOps as *synchronizing HyperOps*. Synchronizing HyperOps wait for the purge messages to be serviced in addition to its input before they execute. The execution of the synchronizing HyperOp implies that the purge messages have been processed and these context memory locations are safe to be reused. To ensure correctness we have to select an appropriate synchronizing HyperOp for each purge request that has been issued. At the end of the function execution, when the context memory locations are released, the locations should be in a consistent state. This in turn means that all messages corresponding to each of these locations should have been serviced before releasing the context memory locations.

The synchronizing point needs to be selected carefully, since not all HyperOps can act as synchronizing HyperOps. For example, a HyperOp on a not-taken path cannot serve as a synchronization point. HyperOps that are certain to be executed after a purge message has been issued are candidates for synchronizing points. Thus, the post dominator of the HyperOp issuing the purge request is a potential candidate for synchronizing. In the absence of loops, the post-dominator is always a safe-point for synchronization. In the case of loops, the synchronizing HyperOp should belong to the same loop as the purge point since the synchronizing HyperOp should execute every time a purge instruction is executed. For example consider the case of purging HyperOp $H_9$ in the example shown in figure 5.1. The immediate post-dominator of HyperOp $H_9$ is $H_{11}$ and it executes once the loop exits. However, the data corresponding to HyperOp $H_9$ has to be cleared for every iteration of the loop. Hence, it is not a candidate. The target of the branch (HyperOp $H_{10}$ in the example) is a candidate in such cases. Please note that, the target of the branch could be a basicblock within the HyperOp. In such a case, all the successors of the HyperOp that can be executed after this branch decision was taken become candidates. Further, since these successors are mutually exclusive, only one of the successors will execute at run time. However, the synchronization point has to be chosen at runtime. In such cases, the decision of purging is put off until the decision point that chooses a successor. In such a case, multiple purge instructions are introduced corresponding to each successor. These instructions are predicated appropriately such that only one of them executes at run time. We refer to such synchronization HyperOps as successor-Synchronization HyperOps to differentiate them from dominator-Synchronization HyperOps. For HyperOps that do not have a successor an additional HyperOp is added that acts like a synchronization point for all such HyperOps. When successor-Synchronization HyperOp is chosen as a synchronization point, it can potentially lead to performance degradation if the execution of the successor-Synchronization HyperOp is on a critical path. Hence, in such scenarios, one of the HyperOp in the iterated post-dominator set of successor-Synchronization HyperOp is chosen as a synchronization point. It is to be noted that, it is possible to choose such a
HyperOp only if both the HyperOps belong to the same loop. Choosing such a HyperOp as a synchronizing point increases the context memory requirement since the line allocated to the HyperOp that is being purged needs to be live until the synchronization HyperOp. In the current version of the compiler, we use immediate post-dominator of the successor-Synchronization point as a synchronizing point when possible. If the immediate post-dominator is not a candidate, we choose the successor of the HyperOp as a synchronization point.

Once we identify the synchronizing point, the purge instruction is annotated with appropriate information so that hardware can inform the synchronizing HyperOp after servicing the request. The number of inputs for the synchronizing HyperOp is suitably adjusted so that the execution of the HyperOp is dependent on receiving a trigger from the HyperOp that is being purged. In the case when a HyperOp has to wait for multiple triggers, they are chained and only the last trigger is sent to the HyperOp. An example of the same is shown in figure \ref{fig:5.2}. In this example, HyperOps $H_3$, $H_4$, $H_6$, $H_7$ should be purged when the execution follows the path $H_1 \rightarrow H_5$. $H_5$ serves as a synchronizing HyperOp for all these purge requests. The triggers are chained and $H_6$ triggers $H_7$ which in turn triggers $H_4$ and finally $H_3$ triggers $H_5$.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure5.2.png}
\caption{Example showing chaining of acknowledgements when path $H_1 \rightarrow H_5$ is taken}
\end{figure}

\subsection{5.2.3 Context Memory Allocation}

As mentioned earlier, it is the responsibility of the compiler to assign appropriate context memory locations for each of the HyperOps. The context memory locations should be allocated such that the HyperOps sharing the same context memory location are not active simultaneously. This problem is similar to that of assigning variables to registers in traditional compilers. In register allocation, the number of registers is fixed. Most appropriate variables are chosen to allocate registers. In the case of assigning context
5.2 Compiler directed resource management

memory locations to HyperOps, each HyperOp should be assigned a context memory location and the objective is to minimize the number of context memory locations used. The key to this problem is to define the liveness of a HyperOp (the technique of solving is similar to the register allocation problem where liveness of a variable is used). To do so, we first define the def and use of a HyperOp.

The def of a HyperOp is the point where the earliest input is produced. Since the HyperOp accumulates inputs in the same context memory location, the generation of the second input is not treated like a re-definition of the context memory location. To determine the earliest input, we consider all the inputs of the HyperOp and determine which input temporally precedes all other inputs. In the absence of such an input the def set contains a set of inputs that together precede all other inputs. Temporal precedence of inputs is defined based on the temporal precedence of basicblocks generating the inputs.

On taken paths, the execution of the HyperOp consumes all the data corresponding to that HyperOp and the context memory location is cleared. Thus, the use of a HyperOp contains the HyperOp itself. On not-taken paths, the HyperOp data is cleared using purge requests, and compiler can reuse the context memory location only after the requests are serviced. In other words, the liveness of the HyperOp extends until the synchronization point that is waiting for the purge to be serviced. Thus, the use of a HyperOp is the set of all synchronizing HyperOps corresponding to this HyperOp and the HyperOp itself. The set of synchronization HyperOps also include the synchronization-HyperOps for purging loop-invariant data. Since the loop-invariant data is valid across iterations of the loop, the context memory locations assigned to such HyperOps can be reused only on loop-exit.

Having defined, def and use of a HyperOp, the problem of finding liveness of a HyperOp is same as finding the liveness of variables. We use the iterated dataflow analysis to find the liveness of HyperOps. Then, we construct a conflict graph, by adding edges between every pair of HyperOps that are live simultaneously. The problem of assigning context memory locations to HyperOps is thus a problem of coloring the conflict graph. We use the algorithm proposed by Coleman and MorAl (1983) for coloring the graph. An example of the entire flow is shown in figure 5.3. For the sake of simplicity, let us consider each basicblock as a HyperOp. The example shows the def and use of each HyperOp and the conflict graph and the context memory locations allocated to each HyperOp.

5.2.4 Delivering data to HyperOps

In the compiler directed scheme, compiler assigns a context memory location for each HyperOp and delivering the data to a HyperOp translates to that of writing data to the specified context memory location. When assigning
these context memory locations, compiler ensures that context memory locations are shared only by HyperOps that are not active simultaneously. For HyperOps executing in a loop, the number of instances of a particular HyperOp that can co-exist is pre-determined by the compiler and the context memory locations are assigned accordingly. Such an assignment is efficient only when applications have deterministic behavior and the parallelism that has to be exploited is known at compile time. In situations where parallelism needs to be unfolded at run time, loops are treated like function calls. On a function call, compiler requests for a specified number of context memory locations and executes the function in the allocated locations. Since these locations are allocated at runtime this method can be used to exploit parallelism that cannot be determined at compile time. This scheme is similar to explicit token store architectures by Papadopoulos and Culler (1998).

Handling loop-invariant data requires special attention since it is delivered once and is used multiple times. Compiler identifies loop-invariant data and indicates it to the hardware. Hardware maintains the count of loop-invariant data corresponding to each context memory location. Once a HyperOp is launched onto the fabric, hardware only invalidates the data that is generated within the loop. Loop invariant data is not invalidated. Once all iterations of a loop complete execution, the loop-invariant data is purged and the context memory location is ready for reuse. Compiler adds instructions
to purge this data appropriately. To achieve this compiler identifies all loops and exit points of each of these loops. It introduces purge instructions at each loop-exit point. Similar to purge instructions, these also should specify the number of loop-invariant data expected. Further, if a loop does not expect any loop-invariant data then it is not required to add purge instructions. Since the context memory location is shared across various instances of the same HyperOp, compiler takes care not to re-allocate this line to any other HyperOps as long as the loop is active.

5.3 Hardware controlled resource management

As discussed so far, compiler directed resource management is efficient for applications that have deterministic behavior. However, for applications where the parallelism needs to be unfolded at run time, compiler directed scheme proves to be a limitation. For such applications hardware controlled resource management can be employed to unfold the parallelism at run-time. In this scheme, hardware needs to allocate resources for the HyperOp at run-time depending on data availability. When a data is delivered for a HyperOp, the hardware first checks if any other data is available for the HyperOp. If an entry for the HyperOp already exists, then the existing entry is updated with the new data. If it is the first data token for the specified HyperOp, it allocates a new context memory location, and updates the context memory location with the data. Hardware has to maintain information about the status of each of these locations and update it as and when required. This introduces additional latencies when exchanging data between HyperOps. In this section, we describe the responsibilities of compiler in the hardware controlled resource management scheme. As discussed earlier most of the tasks are similar to that of the software controlled scheme. The purging/termination of HyperOps, identifying synchronization HyperOps, generating resource requirement matrix are identical in both the schemes. The remaining tasks of data delivery to a HyperOp and handling loop invariant data are discussed in this section.

5.3.1 Data Delivery to a HyperOp

When a HyperOp produces data for another HyperOp, compiler specifies the static identifier of the consumer HyperOp. However, the static identifier alone is insufficient when multiple instances of the same HyperOp are active simultaneously. Instance number is used along with the static HyperOp identifier to uniquely identify the HyperOp. Several instances of a HyperOp could be active because either the HyperOp is a part of loop or the function to which HyperOp belongs is called multiple times. The allocation of instance numbers for each of these situations is handled differently. For a function call,
hardware is responsible for assigning a unique function instance number to
differentiate it from all active functions. The function instances are allocated
by a central manager in the hardware. Loops also can be handled in a similar
manner. However, to reduce the pressure on the central manager and to
reduce the latencies involved in assigning a new function instance, loops are
handled differently. Loops are identified by the iteration index of each of
the loops in the nesting hierarchy of the HyperOp. Since we have limited
number of bits in the hardware for an instance number, there is a limit on
the maximum number nesting levels that can be handled using this method.
Similarly, the number of iterations that can be supported simultaneously is
limited by the number of bits available to store the iteration index of each
loop. When the number of levels of nesting in the application specification is
greater than the number of nesting levels supported by the hardware, it can
either be handled as a function call or by adding additional dependencies so
that only one instance of the loop is active at any point in time, eliminating
the need for instance number. Similarly, dependencies have to be added
to restrict the number of active iterations of every loop. If we employ \( n \)
bits for storing iteration index of each loop, then dependencies have to be
added such that no more than \( 2^n - 1 \) instances are active simultaneously.
Thus, the instance number of a HyperOp contains two parts viz. function
instance number and the loop instance number. Loop instance number
contains the iteration index of each of the loops in the nesting hierarchy to
which it belongs. The example [5.4] shows an example of the instance number
associated with each HyperOp. The example assumes two-levels of nesting.

When data is produced for a HyperOp, we know the instance number
of the producer and the static HyperOp identifier of the consumer. We
need to compute the dynamic instance of consumer HyperOp based on the
instance number of the producer. Compiler provides necessary information
to compute the same. This information includes

- **Producer-consumer loop nesting relation**: Depending on the nesting
  relation of the producer and consumer, the instance number needs to
  be appropriately modified. The relation between the producer and
  consumer can be one of the following:

  - producer and consumer belong to the same loop: In this case, they
    share the same instance number of the parent and only the current
    instance has to be modified based on the dependency distance.

  - producer is the parent loop of the consumer: Since producer
    belongs to the parent loop, it produces data for the very first
    iteration of the consumer. Consumer is at one level lower in the
    nesting hierarchy when compared to producer. Thus, the instance
    number has to be appended by another level of nesting and the
    instance number of the new level is set to zero. When creating the
5.3 Hardware controlled resource management

\[ a = 20 \]

\[
\begin{align*}
&\text{if ( /*condition*/)} \\
&\quad b = a + 30; \\
&\quad \text{else} \\
&\quad \text{for ( } j = 0; j < n; j++) \\
&\quad \quad c += j; \\
&\quad \quad b += c; \\
&\quad d = b; \\
&\end{align*}
\]

\text{HyperOp A}

\text{HyperOp B}

\text{HyperOp C}

\text{HyperOp D}

\text{HyperOp E}

**Figure 5.4:** Code sequence in figure (a) is an example of a nested loop along with the grouping of instructions into HyperOps. As shown in (b) HyperOp A and HyperOp B have only one instance, hence a static tag is sufficient. HyperOp C consumes data produced by HyperOp A multiple times and hence a sticky token is required. HyperOp C and HyperOp D have multiple instances hence a dynamic tag is required. A predicated steer node is necessary for communication between HyperOp D and HyperOp E, since HyperOp D produces the token multiple times. Only the last value should be given as input to HyperOp E.
DFG we ensure that a loop communicates only with its immediate predecessor loop by adding additional steer nodes. An example is shown in figure 5.5. Thus, only one level of appending is required. In a more general case, predecessors that are higher up in the nesting hierarchy can deliver data to a loop. In such a case, partial tag matching has to be used to handle loop invariant data. We discuss this in detail in a later section.

- consumer is the parent loop of the producer: In this case, data is delivered on loop exit to an outer loop. This requires truncating the instance number of the producer by shifting the instance number appropriately. Here, we do not impose a restriction on the number of levels between the producer and the consumer. Thus, we might have to truncate it by arbitrary number of levels and the number of levels needs to be a part of the information.

- **Iteration dependence distance**: It can be clearly seen that the iteration dependence distance can be ‘0’ or ‘1’ for scalar data, since any scalar data produced in a loop iteration has to be consumed in the same iteration or in the next iteration. This is encoded using one bit and serves as a part of compiler information for tag generation. Vector data are not delivered directly to the consumer. Producer stores the data in the Data memory and the consumer loads it from there. Though in some cases it is advantageous to send the vector data directly, we do not currently employ this scheme. When vector data is delivered directly more bits are required to encode the dependence distance.

### 5.3.2 Handling Loop invariant Data

Loop invariant data is produced once and is used for all consumer instances corresponding to this producer. Hence, loop invariant data is stored corresponding to the producer instance number unlike the consumer instance for normal data. The availability of loop-invariant data is checked for each consumer instance of the HyperOp. To check for availability producer instance number has to be derived from the consumer instance number. It is obtained by truncating bits corresponding to the instance number of the consumer as the loop invariant data is always produced outside the loop. In a general case, since the producer can be several levels higher in the nesting hierarchy multiple levels may have to be truncated. Further, HyperOps belonging to multiple levels of nesting can produce loop invariant data. For example, in figure 5.5(a) HyperOp A produces data for HyperOp C. This data is used for all instances of Loop1 and of Loop2. The data produced by HyperOp B is only re-used for all instances of Loop2. A new instance is generated for every instance of Loop1. In this case, partial tag matching has to be
5.4 Summary

Figure 5.5: In the example shown, HyperOp A and HyperOp C communicate, but HyperOp A is not in the immediate parent loop and hence an additional node is added in HyperOp B to ensure each HyperOp communicates only with HyperOps in their parent loop.

employed to identify data produced from multiple levels of nesting. To avoid hardware overhead of partial tag matching we transform the DFG such that only parent-child loops interact.

5.4 Summary

In this section, we discussed the responsibilities of compiler for managing the resources at the orchestrator. Compiler is responsible for providing necessary information for delivery of data, clearing of unused data, and generating resource requirement matrix. The delivery of data depends on the orchestration mechanism. In compiler directed resource management, compiler has responsibility of allocating context memory locations for HyperOps. It is handled by mapping this problem to that of graph coloring problem. We also identify purge points to add instructions to clear the data. We also annotate few HyperOps as synchronizing HyperOps to ensure clear messages are serviced before releasing the context memory location. In hardware managed resource allocation, necessary information is provided along with data to compute consumer instance number. The information includes producer-consumer nesting relation and the dependency distance. The parallelism across loops can be dynamically unfolded in the hardware controlled resource management scheme. When the parallelism that needs to be exploited is known statically, compiler directed scheme could be used to reduce the hard-
ware overhead. The orchestration mechanisms described ensures correctness and provides lightweight synchronization that helps in efficiently exploiting parallelism that exists across HyperOps. Both these schemes support exploitation of parallelism across functions calls, thereby achieving our objective of efficiently exploiting fine-grained task level parallelism and loop-level parallelism.
Chapter 6

Enhancements for performance

So far, we have discussed about correctness aspects of compiling an application specification onto a CGRA employing dataflow execution paradigm. We have discussed the responsibilities of compiler and various mechanisms to handle them. The primary focus was on the correctness aspects of the transformations. In this chapter, we discuss performance related aspects.

This chapter presents four aspects of performance (a) executing loops on fabric (b) exploiting pipeline parallelism (c) exploiting task level parallelism (d) domain customization. As mentioned earlier, loops are not a part of HyperOp. However, for loops that do not have loop-level parallelism it is efficient to execute them on fabric. Further, even for loops that are large and multiple instances cannot be accommodated on the fabric, it is efficient to execute them on the fabric. For example consider the matrix multiplication kernel, the innermost loop accumulates the result, and hence multiple iterations cannot execute in parallel. In such scenarios, the same loop is loaded repeatedly after the previous instance of the loop finishes execution. It will be more efficient to keep the loop resident on the fabric to cut down the launch latencies. In the same example when we consider the outer loop, multiple instances can execute in parallel and it might be efficient to launch multiple instances simultaneously depending on the availability of resources. This is similar to exploiting lightweight task level parallelism. The orchestrator employs a dynamic dataflow scheduling and can handle such kind of parallelism in an elegant way. In such cases, it is the responsibility of compiler to expose the parallelism available in the application. Pipeline parallelism is another important type of parallelism available in streaming applications. For example, in Fast Fourier Transform (FFT) application when large streams of data have to be processed a pipeline can be setup, where
each pipeline-stage performs one stage of the FFT. Domain customization is the key to achieving higher performance in many domains. Thus, the compiler should be able to handle all these enhancements to achieve higher performance. In this section, we discuss the support provided by compiler for each of these enhancements.

Exploiting pipeline parallelism is not completely implemented. In this chapter, we identify various tasks that have to be performed by the compiler to automate the process of exploiting pipeline parallelism. The tasks of exploiting task level parallelism and domain customization are minor optimizations included in this chapter. The major contribution of this chapter is providing support to execute loops on fabric. Static dataflow machines support execution of loops without assuming a flow control mechanism. However, they use acknowledgement tokens between every pair of communicating operations. This means the numbers of acknowledgement tokens are as many as data tokens. Further, this also incurs a long latency in executing operations. One way to reduce the number of acknowledgements is to exchange them between each computation element as opposed to each operation. Yet another level of granularity is the HyperOp itself. In this chapter, we explore both these options. We also propose another approach that does not employ acknowledgements, In that approach loops are transformed by adding additional dependencies to ensure correct execution even in the absence of flow control. The additional dependencies control the generation of tokens for the next iteration of the loop. The tokens for the next iteration are generated only after all the operations in the current iteration have completed, thus ensuring correct execution. In this chapter, we discuss how to determine the minimal number of dependencies that have to be added to ensure correct execution.

6.1 Executing Loops on Fabric

When HyperOps contain loops, each operation can potentially execute more than once. In the absence of a flow control mechanism, there is a possibility of overwriting data before it is consumed since the same location is used to store data belonging to multiple iterations. In the context of our CGRA, since each data token is assigned a unique location, the reuse can happen only across multiple iterations of the loop. Flow control mechanism can be implemented either in software or by adding additional support in the hardware. The software flow-control techniques involve transformation of the loop to make it amenable to execute on a fabric that lacks flow control. Hardware based flow control mechanisms can be broadly classified into two types (a) end-to-end flow control mechanism (b) near-neighbor flow control mechanism. In end-to-end flow control mechanisms, the handshake happens between two communicating entities. Producer has knowledge about the number of
available buffers at the consumer. Consumer sends an acknowledgement to the producer once the data in the buffer is consumed. Producer keeps track of the acknowledgements to estimate the buffer size available at the consumer. Producer stops generating data when the estimated buffer size becomes zero. In the second scheme, the handshake is between near-neighbors and flow control messages are not exchanged directly between communicating entities. Consumer stops accepting the data if the buffers are full and this leads to a backpressure and eventually stops the producer. We discuss each of these techniques in subsequent sections.

In applications that have deterministic computation structure, operations can be scheduled statically honoring all inter-iteration and intra-iteration dependencies. In this model, since all latencies are accounted for at compile time, an additional flow control mechanism is not required. This technique assumes deterministic latencies for all operations including loads and stores. This can be achieved either by pre-loading the data required into the memories or by having a mechanism to stall the operations across all computation elements in the case of unexpected latencies till the request is serviced. This mode of computation is employed in most of the CGRA compilers like DRESC (Mei et al., 2002), PPA (Park et al. 2009). This mode of execution is not in the scope of this thesis. However, the efficiency of such a scheme for linear algebra kernels in the context of our CGRA has been discussed by Biswas et al. (2010).

6.1.1 Well behaved loops and Transforming to well behaved loops

As discussed earlier, we require a flow control mechanism when a location is reused for multiple data items. Further, flow-control mechanism is not required when it can be asserted that the data assigned to a location is consumed before a new data is written into the location. In other words, every reuse of the location asserts that the earlier data has been consumed and is not required anymore. This can be trivially ensured, if the operation that is producing data for a re-used location is dependent on the consumer of the data at that location. For example in figure 6.1(a) consider the delivery of data from operation \( C \) to operation \( A \). Operation \( C \) is dependent on operation \( A \), and operation \( C \) can execute only after operation \( A \) has consumed its data. Hence, in this case it is safe to deliver data to operation \( A \). However, when delivering data to operation \( D \) from operation \( A \), the execution of operation \( A \) is not dependent on execution of operation \( D \) and hence it can potentially lead to buffer overrun. It is to be noted that the buffer overrun happens because of data produced from multiple iterations of the loop. Thus by controlling the delivery of data at loop boundaries we can ensure that buffer overrun does not occur. If we ensure that the data is delivered only after all operations in the loop have finished execution, we can eliminate the case of
buffer overrun. However, this condition is stricter than necessary condition to ensure correct execution. We can relax it further by delivering the data after all the operations that are affected by the arrival of new data have finished execution. In the earlier example, for the data delivery on the backedge from \( G \rightarrow G \), it is sufficient to wait for operations \( G, H, I \) have finished execution. The delivery of the data for the new iteration does not affect the rest of the operations in the loop. For the data delivery on the backedge from \( C \rightarrow (A,G) \), it needs to wait for all the operations in the loop. The loop shown in the example can be transformed by adding additional dependencies (as shown in figure 6.1(b)) such that the data delivery happens only after the affected operations have finished execution. To state it more formally, we define a well-behaved loop as follows:

**Definition:** A well-behaved loop is a loop where data delivery from each source delivering data on a backedge is well-behaved. A data delivery from a source delivering data on a backedge is well-behaved if all paths from each of the destinations of the source terminate at the source.

Intuitively, we can see that such loops do not require a flow control mechanism to execute correctly.

**Claim 6.1.1.** Well-behaved loops do not require flow control mechanism to ensure correct execution

**Proof.** As discussed earlier, a flow control mechanism is only required for delivering data across iterations. A well-behaved loop, ensures that all

![Figure 6.1](image-url)
operations that will be re-evaluated on the delivery of a particular data have finished execution before delivering that data token. This is ensured since the source operation that is delivering data to the next iteration is dependent on all the operations that will be re-evaluated in the next iteration. Hence when delivering the data all the data buffers are free and does not lead to a buffer overrun. Hence, an additional flow control mechanism is not required for correct execution.

In this scheme, all loops are transformed to well-behaved loops so that they can execute safely on the fabric. The loops are transformed into well-behaved loops by adding additional dependencies and making each backedge well behaved. To do so, we identify all the operations that are reachable from the destination of the backedge and do not terminate at the source of backedge. We refer to these operations as *leaf-operations*. Once these operations are identified, we add additional dependencies from each of those operations to the source of the backedge. The number of additional nodes added depends on the number of paths that do not terminate at the source of the backedge. Another way of accomplishing the same without modifying DFG involves grouping together all *leaf-operations* and the source of the backedge into one computation element. If the computation element ensures that all operations of a particular iteration have executed before starting the next iteration, then it has the same effect as adding additional dependencies. Further, unlike the earlier scheme where data delivery is predicated, here data is delivered and the execution is predicated. This allows overlap of the data delivery with the execution of the current iteration of the loop. An example of the same is shown in figure 6.1(c). Please note that, this scheme is applicable only if all the leaf-operations can be accommodated in one computation element.

When loops with multiple levels of nesting are executed on the fabric, loop invariant data needs to be handled in addition to the flow control mechanism required for execution of loops. Loop invariant data of the inner loop has to be cleared for each iteration of the outer loop. It is difficult and inefficient to implement such a scheme in software. The technique of transforming loops to well-behaved loops only addresses the problem of flow control and does not handle the loop-invariant data. Hence, we restrict this technique only to innermost loops.

### 6.1.2 End-to-end flow control mechanism

In end-to-end flow control mechanism, the information is exchanged between the producer and consumer of data. The producer has the knowledge of the size of buffers available at the consumer and is allowed to send as many data tokens as the number of buffers without waiting for any external signal. The consumer is responsible for informing the producer once the data in the
buffer has been consumed. Using this information producer keeps track of the buffers available at the consumer. When the estimated number of buffers at the consumer is zero, producer stalls itself. When there is only one buffer available at the consumer, it introduces a round-trip delay for restarting the producer. Further, in the context of executing HyperOps on the fabric it is important to determine the granularity of the communicating entities. This scheme when implemented at instruction level granularity assuming a buffer size of one translates to that of static dataflow execution model. As discussed earlier, static dataflow scheme requires as many acknowledgement tokens as data tokens. Further, it is expensive to maintain buffers for each instruction and it results in underutilization of the buffers in many cases.

The next level of granularity is the level of pHyperOp. As mentioned earlier, pHyperOp is a partition within a HyperOp that can be mapped onto a computation element. In this scheme, acknowledgements are exchanged at the level of computation elements. Since each computation element consists of multiple operations, the overhead of acknowledgements is amortized over the set of operations in the computation element. Each computation element is responsible for executing all the operations belonging to the current iteration, before starting the next iteration. In addition, it also needs to wait for a free space in all the computation elements that receive data from this computation element. This scheme is efficient if HyperOp is partitioned horizontally i.e. consisting of multiple stages without a feedback from different stages of pipeline. In other words, the HyperOp should be partitioned as a feed-forward pipeline. An example of the same is shown in figure 6.2(a). It shows three stages of pipeline with feedback only from the first stage to itself. This kind of interaction typically occurs when there is a complex computation in the loop and the result of the loop is not fed-back to the next iteration. In other words, the computation is repeated over large sets of data without dependences across each set of data. For such kinds of loops, loop-level parallelism can be exploited by executing multiple instances of the loop in parallel. Hence, one possibility is to assign the responsibility of scheduling them to the orchestrator. Another possibility is to execute the loop in a pipelined manner on the fabric. In such cases, implementing flow control at the level of pHyperOps is efficient as it allows multiple pHyperOps to execute in parallel on different sets of data, effectively implementing a pipeline. Near-neighbor flow control is more suitable for pipeline parallelism.

When there is a feedback from the computation i.e. the result produced at the end of iteration is used in the beginning of the next iteration (as shown in figure 6.2(b)), having a flow control mechanism at the level of pHyperOp does not improve performance. Further, merge points require

---

1It is possible to pipeline the loops with dependencies also, but it is more involved than the case with no dependences. It involves looking at the outer loop and pipelining it appropriately such that all dependences are honored.
special processing so that the operands are received in correct order from multiple iterations of the pHyperOps. This is discussed in detail in the section 6.1.3. For such cases, we implement a flow control mechanism at the granularity of the HyperOp. In this scheme, a HyperOp executes to completion before starting the next iteration. Since the context of HyperOp is distributed across multiple computation elements, it is non-trivial task to determine the completion of iteration. To address this problem, we identify one computation element as a leader computation element that keeps track of the iteration complete messages of each of the computation element that belongs to a HyperOp. Each computation element is programmed to inform the leader computation element once it executes one iteration to completion. Leader computation element is responsible for waiting for the correct number of acknowledgments and then issue messages to restart each of the computation elements that belong to the HyperOp. It is also responsible for invalidating the loop-invariant data on a loop exit. The result of loop exit condition is informed to the leader computation element. Depending on the result, it invalidates the loop-invariant data. An example is shown in the figure 6.3. In this scheme, buffers can be employed to hide the delay due to the exchange of control information. For most of the applications, it is sufficient to employ two buffers to hide all control latencies. In the presence of buffers, tags should be employed to identify data from different iterations.
Unlike the tags employed at the orchestrator these tags are less complex to implement, as we do not allow overlap of multiple iterations of the outer loop. However, we do not implement such a scheme on fabric.

6.1.3 Near-neighbor flow control mechanism

An end-to-end flow control scheme introduces round-trip delays for exchanging data between two communicating entities. Point-to-point networks and programmable networks enable single cycle communication between two communicating entities. Hence, for such networks end-to-end flow control mechanism is not a bottleneck. However, the presence of NoC on the fabric can introduce arbitrary delays depending on the path length and other dynamic factors like traffic present on the path. Further, NoC employs routers that ensure flow control between routers that are directly connected to them. This flow control mechanism can be used to implement an end-to-end flow control. To reduce the delay of acknowledgements the flow control provided by the routers can be used to implement an end-to-end flow control. In this scheme, the consumer stops accepting the data when its buffers are full. Thus, the router cannot deliver any more packets to the consumer and hence blocks the previous router on the path. This builds up backpressure and eventually stops the producer from producing the data.
6.1 Executing Loops on Fabric

if ( i % 2 == 0 )
    a1 = b * c
else
    d = e * f
    g = d * h
    a2 = g/23
    a = phi (a1, a2)
store a, addr

Figure 6.4: An example code sequence and one possible mapping of the code onto the fabric. With the mapping shown in this figure it leads to an incorrect execution (a) shows the snapshot when the CE D has finished the first iteration (b) shows the snapshot when the data is delivered to CE C from CE B for the first iteration. By this time, CE D would have finished second iteration. (c) shows the snapshot when the third iteration data from CE B is ready at CE C, but second iteration data is yet to be received.

In this scheme, the flow control happens between each pair of computation elements. Each computation element executes the current iteration of the loop to completion and starts the next iteration. However, additional information is required to maintain the correct order of tokens at merge points. For example, consider a hypothetical code sequence and a possible mapping of it onto the computation elements as shown in figure 6.4. In this scheme Compute Element A (CE-A) and Compute Element B (CE-B) provide data at the merge point for every alternate iteration. Since CE-B has a long latency operation, CE-A produces data faster than CE-B and hence multiple data tokens of CE-A can be potentially accepted before the data token from CE-B is available. Figure 6.4 shows snapshots of the execution.
at three different time instances. Figure 6.4(a) depicts the first iteration of the loop. Since $CE-D$ contains only one instruction, it proceeds with other iterations as $CE-A$ is executing instructions belonging to first iteration. $CE-B$ also has only one instruction and it also can execute multiple iterations before $CE-A$ can finish one iteration. Due to this, data token $a$ is not delivered in the correct order at $CE-C$. To avoid such situations each data token can be tagged appropriately. However, in a general case it is difficult to determine the number of bits required for correct execution. In the example mentioned earlier, the number of bits required depends on the relative latencies of the execution of operations in $CE-A$ vs. $CE-B$. Further, in the above example if the condition is data-dependent it is not possible to determine the number of bits required for correct execution. If the partitioning is changed and all the data received by a merge point is generated by a single computation element then such a problem does not arise. In the same example, if we partition the code sequence differently as shown in figure 6.5 such a situation does not arise. In other words, if data is delivered from the same computation element for every iteration of the loop and the delivery between each pair of elements is ensured to be in-order, then a single bit is sufficient to ensure flow control.

```
if ( i % 2 == 0 )
    a1 = b *c
else
    d = e * f
    g = d * h
    a2 = g/23
    a = phi (a1, a2)
store a, addr
    d = e * f
    a1 = b *c i % 2
    store a, addr
    a1 = b *c
    a2 = g/23
    g = d * h
    d = e * f
```

Figure 6.5: An example mapping of the code onto fabric such that every pair of communicating entities transfer data for every iteration of the loop

Further, when the consumer is not in a position to accept data, the data is blocked at the router. The router will not be able to service requests in that direction. If the same path is required to deliver data for an earlier iteration, it will lead to a deadlock. Adaptive routing can reduce the probability of a deadlock by choosing an alternate path but cannot eliminate the possibility of deadlock. As an example, consider the mapping shown in figure 6.5 the link from $A$ to $C$ is shared between the communications for $A$ to $C$ and $B$ to $C$, and since the computation in $A$ happens faster than communication from $B$, it leads to a deadlock. Thus, the mapping step of the compilation should ensure that there are no such situations. In a naive scheme, compiler
can map such that communication paths between all pairs of communicating entities are non-overlapping. However, it restricts the mapping step more than required. Ensuring non-overlapping paths is sufficient to eliminate the possibility of a deadlock. However, it is not required for ensuring deadlock free execution. Currently this feature is not implemented in the compiler and we expect manual mapping of partitions onto the fabric. This scheme of flow control was employed for exploiting pipeline parallelism as described in Alle et al. (2009).

6.2 Other Enhancements

So far, we have discussed techniques to compile applications to a CGRA employing dataflow execution paradigm. These techniques have to be combined with several other traditional analyses to enhance performance. In this section, we discuss few important areas where additional analysis is required to improve performance. The compiler is not currently equipped to use this analysis efficiently. However, it is capable of accepting user hints and generates necessary code.

6.2.1 Pipelining Loops on Fabric

Pipeline parallelism is especially useful in streaming applications, where the same set of operations has to be executed on different sets of data. Further, in streaming applications data is not available apriori. Thus, spawning of multiple threads is not always beneficial. Further, threads increase the pressure on load-store units. Pipeline parallelism is the most suitable form of parallelism in such scenarios. To exploit pipeline parallelism, the operations are partitioned into multiple stages akin to the design of traditional pipeline. The first stage of the pipeline receives the data from the data stream and passes the processed data directly to the next stage. Since the data is delivered directly without accessing the load-store unit, it eases the pressure on the LSU. Depending on the number of stages employed, pipeline parallelism can improve performance significantly. In non-streaming applications that work on large data sets, pipeline parallelism can prove to be more efficient than task level parallelism since the data produced by one stage can be streamed to an appropriate stage without accessing LSU.

To implement pipeline parallelism we need flow control mechanism for communications taking place between different stages of the pipeline. End-to-end flow control mechanism described earlier in this section is not suitable for pipeline parallelism. Near-neighbor flow control is more efficient since it does not incur round-trip delays unlike the earlier scheme. Since flow control is required only for data exchanged between different stages and pipelines typically have feed-forward dependencies, it will be easier to find
paths such that it does not lead to a deadlock. In most of the cases, it is easy to ensure non-overlapping paths between the communicating stages. As discussed earlier, ensuring non-overlapping paths between every pair of communicating entities is sufficient to ensure deadlock-free execution.

To automate the task of identifying pipeline parallelism compiler is responsible for the following tasks:

- Compiler needs to identify suitable portions of the applications for pipeline parallelism. For streaming applications user can specify the core loops that process streaming data. For other applications that process large sets of data, suitable code sequences can be identified based on the number of iterations of the loop and the amount of processing required in the loop.

- Partition the task into multiple stages such that the execution time is balanced in each of the stages. When the computation assigned to a particular stage contains loops, the execution time needs to be estimated by considering the number of iterations of each of these loops. Further, in some applications it may be required to unroll loops and then partition them to multiple stages to obtain an efficient implementation. To perform such an analysis, compiler requires information about latencies of various operations and the communication delays on the fabric.

- Data exchanged between various stages should be delivered directly on the fabric instead of going to the LSU. In many cases, the data that is exchanged across various stages is vector data. To analyze this data, compiler needs to analyze the array access patterns based on loop indices. Techniques like Polyhedral analysis can be employed to obtain accurate information (Grosser et al., 2011a). This technique is applicable for loops that fall in the class of static affine nested loops. Most of the loops in computational workloads belong to this class. Hence, this technique is very efficient in determining the dependencies and hence determining the data that can be streamed to various stages of the pipeline.

Currently, compiler is not equipped to handle the earlier mentioned tasks. Compiler expects programmer to specify the loop that needs to be pipelined, various stages in the pipeline and the data that has to be streamed across stages. When this information is available compiler is capable of generating necessary code, to exploit pipeline parallelism.

6.2.2 Task/Thread Level Parallelism

TLP is another important parallelism that needs to be exploited to obtain good performance. Dataflow execution is well suited for exploiting TLP.
6.3 Summary

as synchronization is implicit in dataflow execution paradigm. The common mechanism to exploit TLP is at the level of threads specified by the programmer. Since our CGRA supports lightweight synchronization, TLP can be exploited at a finer granularity in addition to the parallelism across threads. When multiple code sequences within an application/thread can execute in parallel, compiler is responsible for grouping them into different HyperOps. Since these HyperOps do not have dependencies between them, hardware can execute them in parallel depending on the availability of resources. Function calls are another important source of TLP. When function calls can execute in parallel then precedence edges should not be added between them. This allows hardware to execute them in parallel. Inter procedural analysis is required to determine if functions can execute in parallel. Currently the compiler does not include inter-procedural analysis and the programmer is expected to specify the parallelism available between function calls.

6.2.3 Domain specific FUs

As shown in various studies domain specialization is the key to achieve throughput demanded by high-end applications (Borkar and Chien, 2011). To give a simple example, butterfly unit plays a key role in improving performance of FFT application. Intrinsics enable the compiler to generate code that employs custom functional units available on the fabric. The custom FUs are specified as function calls in the C specification. These function calls are replaced by operations corresponding to the custom FU as directed by the user.

6.3 Summary

This chapter presents the performance enhancements supported by the compiler. This chapter discusses the execution of loops on the reconfigurable fabric. Executing loops on a fabric that employs dataflow execution paradigm without support for tagging the data is the contribution of this chapter. The reconfigurable fabric lacks the support for flow control. In the earlier literature, loops were mapped onto the fabric by assuming deterministic latencies. In this chapter, we provide a technique to execute loops on the fabric even in the presence of non-deterministic latencies. We then discuss executing nested loops on the fabric by assuming little hardware support. We discuss end-to-end flow control and near-neighbor flow control mechanisms and indicate the suitability of these mechanisms when exploiting different kinds of parallelism. Near-neighbor flow control is most suitable for exploiting pipeline parallelism. End-to-end flow control is suitable for executing nested loops on the fabric. We also briefly discussed other traditional analysis
required to generate better code.
Chapter 7

Results

In this chapter, we evaluate the application kernels from two different domains. These applications are targeted to different instances of our CGRA. This is achieved by changing the configurable parameters of the compiler. This shows the flexibility of our compiler. We present the results of various compiler optimizations that are completely supported by the compiler. Though the compiler supports pipeline parallelism, it is not discussed in this section since the techniques are not completely automated and require considerable manual intervention. To exploit pipeline parallelism available in the application, the dependencies across the stages of the pipeline, the mapping of each of these stages onto the fabric needs to be specified manually. The results of manual mapping of QR factorization kernel are discussed by Biswas et al. (2010). A manual mapping of FFT on our CGRA has been presented by us in Alle et al. (2009) and Fell et al. (2009). In both these attempts, we obtained high throughput comparable to that of Application Specific Integrated Circuit (ASIC) implementation. We present these results also in this section. The compiler can be enhanced to automate this by using traditional analysis like dependency analysis and polyhedral loop analysis.

We first describe the applications chosen for performance evaluation and describe the experimental framework. Subsequently, we present the results and analysis of the results that provides more insight into the framework.

7.1 Applications

We consider applications from two different domains viz. cryptography, and linear algebra. Cryptographic protocols are used mainly for message encryption/decryption and message authentication. Cryptographic algorithms employed for the above tasks can be broadly classified into two types: symmetric key algorithms and public key algorithms. In symmetric key algorithms, the
communicating parties share a common key. This key is used for message encryption/decryption and to authenticate messages. Advanced Encryption Standard (AES) is a symmetric-key algorithm used in many cryptographic standards. In public key cryptography, each user has two keys viz. private key and public key. For ensuring secure communication, the sender uses the public key to encrypt a message and the user can decrypt using his/her private key. To authenticate messages, the user signs it using his/her private key and receiver uses the public key to authenticate the messages. Elliptic Curve Diffie-Helman (ECDH) is a key agreement protocol based on public key-private key mechanism. It is used for establishing a shared key between two communicating parties. Elliptic Curve Digital Signing Algorithm is another example of public key-private key algorithm used for message authentication. In any public key-private key based mechanisms it is essential to ensure that it is infeasible to generate private key given the public key and vice versa. In Elliptic Curve Cryptography algorithms, this is due to the computational infeasibility of Elliptic Curve Discrete Logarithm Problem. Elliptic Curve Cryptography algorithms involve computation of \(nP\) where \(n\) is a scalar and \(P\) is a point on the elliptic curve. This is achieved by using Elliptic Curve Point Addition (ECPA) and Elliptic Curve Point Doubling (ECPD). These two kernels form the basic building blocks of Elliptic Curve Cryptography algorithms.

For the purpose of evaluation, we choose 128-bit AES encryption and decryption algorithms, Secure Hashing Algorithm-1 (SHA-1) (National Institute of Standards and Technology, 2002), ECPA and ECPD. In addition to the cryptography applications, we also include Cyclic Redundancy Check (CRC), Inverse Discrete cosine transform (IDCT) and sobel edge detection algorithm into this set. These applications are included to demonstrate the flexibility of the compiler and architecture to execute different kinds of applications on the same hardware. All cryptography kernels (ECPA, ECPD, AES, SHA-1), have a few load instructions followed by a large computation i.e. the computation to load instruction ratio is very high. These are not load limited and have complex computation with a lot of parallelism. CRC is a small kernel which is load-store intensive with very little computation. CRC is executed for a 256-byte block for the purpose of our experiments. Sobel Edge detection is similar to CRC in terms of the load-store to compute ratio. Sobel Edge detection algorithm is a larger kernel than CRC. This edge detection algorithm is executed on an image of size \(382 \times 204\). IDCT has a moderate compute to load-store instruction ratio. This is computed for a block of size \(8 \times 8\).

From the linear algebra domain, we look at matrix kernels like matrix-matrix multiplication, LU decomposition, QR factorization using Givens rotation. Apart from these, we also include radix-2 FFT and Magnetic Resonance Imaging-Q (MRI-Q). The matrix multiply kernel has a simple nested loop structure with more load instructions than compute instructions. The
7.2 Experimental Framework

The block diagram in figure 7.1 shows the reconfigurable fabric connected to the orchestrator through the peripheral routers. The reconfigurable fabric comprises a set of compute units and routers. The compute unit is capable of storing a fixed number of instructions. Along with each instruction, there is storage available for its three data operands and predicate. When all operands of an instruction are available, it is ready for execution. The ALU within each compute unit, comprises several FU. The type of FU employed in a computation unit depends on the domain for which the reconfigurable fabric has been customized. In this thesis, we experiment with two different domain-customizations. One of them is for cryptography. The other domain customization is done for supporting floating-point applications (mostly related with linear algebra). The list of FU supported in these domain-specific fabric are listed in table 7.1. Alongside each FU the latency is also recorded. The floating-point fabric employs two types of compute units; one compute unit for integer operations and another compute unit for floating
point operations. For cryptography domain, we employ two special purpose functional units specifically meant for accelerating cryptography applications. The functionality of these two units are:

- **Field Multiplication & Barrett Reduction**: A single unit is used to implement Galois Field Multiplication over binary fields and reduction. The design of this unit has been presented by [Das et al., 2011]. The unit supports 8-bit, 16-bit, and 32-bit width operands. In the case of 8-bit and 16-bit operations, the unit supports vectored execution mode. Each operation takes 3 clock cycles to execute. These operations are extensively used in the context of Elliptic Curve Cryptography.

- **Field Squarer**: This is another Galois Field operation. This performs the squaring operation of a number in Galois field. The result is in unreduced form and executes in a single clock cycle.

Each of the Load/Store Units connected to the peripheral router is connected to a data memory bank of size 128KB as shown in figure 7.1. The maximum permitted address space per bank is $2^{29}$ bytes. The bank size was chosen to be 128KB for purposes of simulation. An equal amount of memory per bank is allocated to the instruction memory. These are connected to the instruction and data transfer unit.

We target two variants of our CGRA. The details of these variants are tabulated in table 7.2. In platform variant I, hardware handles the context memory management whereas in platform variant II, the context memory management is the responsibility of the compiler. In both these variants,
Table 7.1: List of all FUs and the latency in clock cycles for the Cryptography fabric and floating-point fabric.

<table>
<thead>
<tr>
<th>Type of CE</th>
<th>FU</th>
<th>Latency in cycles</th>
<th>Type of CE</th>
<th>FU</th>
<th>Latency in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cryptography Fabric</td>
<td></td>
<td></td>
<td>Floating Point Fabric</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Add &amp; Compare unit</td>
<td>1</td>
<td>I</td>
<td>Add &amp; Compare unit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Integer Multiply</td>
<td>6</td>
<td></td>
<td>Integer Multiply</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Bitwise and Logical Operations</td>
<td>1</td>
<td></td>
<td>Bitwise and Logical Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Data Transport Operations</td>
<td>1</td>
<td></td>
<td>Data Transport Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Load-Store Operations</td>
<td>1</td>
<td></td>
<td>Load-Store Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Shift Operations</td>
<td>3</td>
<td></td>
<td>Shift Operations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Field Multiplication &amp; Barrett Reduction</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Field Squarer</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Floating Point Add &amp; Subtract</td>
<td>9</td>
<td>II</td>
<td>Floating Point Add &amp; Subtract</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Floating Point Multiply</td>
<td>6</td>
<td></td>
<td>Floating Point Multiply</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Floating Point Divide</td>
<td>21</td>
<td></td>
<td>Floating Point Divide</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Floating Point Sin &amp; Cos</td>
<td>29</td>
<td></td>
<td>Floating Point Sin &amp; Cos</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Floating Point Square Root</td>
<td>36</td>
<td></td>
<td>Floating Point Square Root</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Integer to/from Floating Point type conversion</td>
<td>1</td>
<td></td>
<td>Integer to/from Floating Point type conversion</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 7.2: Block diagram showing the pipeline stages of the pipeline

Table 7.2: Details of the two platform variants used in our experiments

<table>
<thead>
<tr>
<th>Component</th>
<th>Platform Variant I</th>
<th>Platform Variant II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orchestrator</td>
<td>Hardware controlled resource management (refer 5.3)</td>
<td>compiler directed resource management (refer 5.2)</td>
</tr>
<tr>
<td>Size of context memory</td>
<td>64KB</td>
<td>64KB</td>
</tr>
<tr>
<td>Type of CE</td>
<td>Priority Encoder to select multiple ready operations</td>
<td>Schedule determined at compile time</td>
</tr>
<tr>
<td></td>
<td>4 cycles delay between dependent operations</td>
<td>2 cycle delay between dependent operations</td>
</tr>
<tr>
<td>Fabric Size</td>
<td>$5 \times 6$</td>
<td>$5 \times 6$</td>
</tr>
<tr>
<td>Data Memory</td>
<td>128KB</td>
<td>128KB</td>
</tr>
</tbody>
</table>
7.3 Analysis of static dataflow graph properties

Orchestrator context memory of 4KB is used. This can accommodate 64 contexts each with 16 data operands where size of each operand is 32 bits. In the platform variant I, where we employ a hardware controlled orchestration, along with the context memory a look up table of size 80Kbits is used. This look-up table is sufficient to implement a 4-way look up table with a 10-bit HyperOp identifier. We also provide a loop-invariant store of size 64KB. The computation element employed in this variant uses a priority encoder to choose between multiple ready operations. It introduces a latency of 4 cycles between executions of two dependent operations because of the absence of bypass. In the platform variant II, the computation element follows a static schedule as determined by the compiler. It does not employ a priority encoder and executes the instructions in-order. To ensure deadlock free execution, the instructions should be scheduled in topological order. We use a force directed scheduling algorithm to determine the schedule. In this computation element, the latency of execution between two dependent operations is reduced to two cycles. The pipeline stages of this compute unit is shown in figure 7.2.

In both these variants we use a $5 \times 6$ array of computation elements as execution fabric and 6 peripheral routers to connect fabric to the orchestrator. The entire implementation was done using Bluespec System Verilog. Hardware designs described in Bluespec can be translated automatically into C++ or Verilog. The verilog generated by the bluespec compiler was used to synthesize the hardware and simulations were performed using C++ simulator generated by the bluespec compiler.

7.3 Analysis of static dataflow graph properties

The effectiveness of dataflow graph creation can be measured in terms of the number of non-compute instructions (instructions that do not perform any computation) added in addition to the compute operations in a dataflow graph. When transforming an imperative language specification to a dataflow graph, the control information has to be transformed into dataflow. In addition to this, the notion of order that is inherent in an imperative specification needs to be captured explicitly. The overhead due to the addition of these non-compute instructions is an important concern when building the dataflow graphs. Dataflow graphs typically have a higher overhead of control instructions when compared to the imperative specification. An efficient compiler has to minimize these overheads. To measure the efficiency of our compiler, we present the percentage of control instructions in the dataflow graph. We also plot the percentage of control overhead in the application specification to measure the relative increase in the same. The control overhead in the imperative language is due to branch instructions. In dataflow graph, the control overhead is due to the addition of predicated steer nodes for data
delivery, delivering data to the merge points and additional operations to ensure the correct order between memory operations. In addition to these, data propagate nodes are added for delivering data from a load when it is consumed by multiple destinations. The LSU can deliver data to only one instruction. Hence, when multiple destinations exist for a load an additional node is added to propagate the result. Figure 7.3 shows the percentage of control instructions in the dataflow graph. The control overhead in the dataflow graph varies widely from 3-38%. Earlier study by Petersen et al. (2006) show an overhead of 30-45% of total operations. The studies were performed for general-purpose processors that may have higher control instructions and lesser parallelism. Since the additional operations added in the dataflow graph is highly subjective to the application characteristics we cannot compare them directly. However, they give an approximate indication of control overhead in dataflow graphs. We compare the control overhead in dataflow graph against the control overhead inherent in the application specification. On average, the control overhead doubles in the dataflow graph when compared to the control overhead in the imperative specification. Such a comparison is not available in the results reported by Petersen et al. (2006). For applications Advanced Encryption Standard Encryption (AES-E), Advanced Encryption Standard Decryption (AES-D), ECPA, ECPD, IDCT, there is very little control and hence we see minimal overhead in the dataflow graph. However, we observe that the control overhead is more than double when compared to the percentage of branch instructions. As mentioned earlier, control overhead in dataflow graphs also include the data propagate instructions added to handle multiple destinations of a load. In these applications, this contributes to the increase in the number of non-compute nodes in the dataflow graph. The relatively large increase in ECPA is due to the presence of more number of nodes to propagate the output of a load instruction. In SHA-1 and matrix multiplication we see a higher overhead due to the memory parallelism available in these applications. For these applications, we can notice a corresponding increase in the percentage of memory precedence edges as shown in figure 7.4. Since there are accesses to different arrays that do not alias with each other, we incur a higher overhead in adding more precedence edges. For FFT application, we see close to 3× increase in the control overhead when compared to the imperative specification. This is due to the memory parallelism available in this application. However, for this application we do not see a large increase in the percentage of memory precedence edges. In this applications memory dependencies had to be passed through multiple levels of nesting and $\phi$ nodes are employed to merge the precedence edges at join points. The $\phi$ nodes receive these precedence edges as data input and hence we see a larger increase in the number of vertices when compared to the number of edges.

Along with the number of operations, we also measure the non-data edges in the dataflow graph. Each edge corresponds to a transport and it
7.3 Analysis of static dataflow graph properties

Figure 7.3: Graph showing the control overhead for various applications

gives an indication of the number of transports required. We break down
the edges into data edges, loop-invariant data edges, precedence edges and
predicate edges. Data edges and loop-invariant data edges transport the
actual data and precedence edges and predicate edges transport control
information. Precedence edges are added to maintain the correct order
between operations and predicate edges transform control information to
dataflow. The percentage of different kinds of edges is shown in figure 7.4.
The percentage of control edges varies from 5-40%. As in the earlier case, we
see higher overhead for most of the control-intensive applications. IDCT on
the other hand, has a higher overhead in spite of being less control intensive.
This is due to the large number of predicate edges because of the parallelism
available in the application.

In a dataflow execution paradigm, we deliver data to the consumer unlike
the control flow execution where the consumer fetches it from the registers.
Thus, in dataflow execution paradigm the data has to be delivered to multiple
consumers. To measure the data duplication due to this we present the
number of destinations in the dataflow graph for various applications in
figure 7.5. For all of the applications we see a peak at one, suggesting that
the data is consumed locally many times. However, we also see few data
operands consumed by as many as 40-45 consumers in ECPA and ECPD
applications. In other applications the maximum number of destinations
varies from 5-15. The percentage of such kind of nodes is small and hence
we do not expect it to have a large impact on performance.

So far, we presented the analysis of the control overhead of the dataflow
Figure 7.4: Graph showing the break-up of different kinds of edges in the dataflow graph.

Figure 7.5: Plot showing the number destinations in the dataflow graph.
7.3 Analysis of static dataflow graph properties

Now, we look at the control overhead of the HyperOp. Once the dataflow graph is partitioned into multiple HyperOps, additional processing is done to make them suitable for execution on the fabric. In our CGRA, currently each operation can deliver data to at most three destinations. We add output propagate nodes when the number of destinations is more than three. We employ predicated steer nodes to deliver data within a HyperOp. Further, we also add instructions to transfer data to other HyperOps. Thus, HyperOps have a higher control overhead when compared to the dataflow graph. The percentage of different kinds of operations in each application is shown in figure 7.6. The control operations shown in the figure correspond to synchronizing operations required for implementing memory barriers for a function call. For linear algebra kernels that contain multiple levels of nesting, we see a relatively higher overhead. This is because we do not include loops when forming HyperOps. This leads to smaller HyperOps and hence larger overhead for data transfer across HyperOps. MRI-Q, CRC have lesser number of loops and hence relatively lesser overhead. In FFT, we see a significant overhead because of the memory parallelism available in the application.

![Figure 7.6: Plot showing break-up of different kind of nodes after forming HyperOps](image)

Figure 7.7 shows the percentage of different kind of edges after creation of HyperOps. We see a similar trend observed in figure 7.4. We notice a reduction in the percentage of non-data edges. This is due to the following reasons:

- The predicates generated outside the HyperOp are delivered to the
HyperOp instead of delivering to individual operations. Since all the operations in the HyperOp are governed by this predicate, it is not necessary to add predicate edges from the input port of the HyperOp.

- Similarly, precedence edges that are generated outside the HyperOp are not required to be delivered to the operations within the HyperOp.

The reduction in the number of predicate transfers helps reduce HyperOp launch latencies.

![Figure 7.7: Plot showing the break-up of different kinds of edges after forming HyperOps](image)

### 7.4 Analysis of execution time

We measure the performance of the application for various hardware configurations. These hardware configurations can be considered as an architectural design space exploration. These hardware configurations are the instances of the target architecture described in section 2.3. The orchestration model and the synchronization model are based on dataflow execution paradigm in all hardware configurations we have evaluated. The compiler remains the same across these hardware configurations other than changes to configurable parameters. The various hardware configurations are listed below:
7.4 Analysis of execution time

- Platform Variant I: In this instance of the CGRA, the context memory is managed by the hardware as described in section 5.3. The instruction selection mechanism within a HyperOp is also a responsibility of the hardware. A priority encoder is employed to select an instruction when multiple instructions are ready simultaneously. A NoC based interconnection network is employed on the reconfigurable fabric.

- Platform Variant II: In this instance, the compiler controls the context memory. The instruction scheduling is also a responsibility of the compiler. We employ a Force Directed Scheduling (FDS) to schedule these operations. The hardware selects the instructions in the order specified by the compiler. This eliminates the need for the priority encoder at the computation element. The other parameters are same as Platform Variant I.

Apart from these, we also vary the size of the instruction buffer that is available at the computation element, in our experiments. Platform variant I and platform variant II differ in both the orchestration mechanism and the instruction selection mechanism change. We do not try the other combinations like (i) hardware controlled context memory + instruction scheduling by the compiler (ii) compiler directed context memory allocation + priority encoder based instruction selection. This is due to the high engineering effort involved. The instruction selection mechanism has an effect on the time spent on the reconfigurable fabric and the orchestration mechanism employed has an effect on the time spent on the inter-HyperOp communication. However, it is difficult to isolate the effect of these two parameters on the performance of the application due to the overlap of the inter HyperOp communication and fabric execution.

The first set of experiments use platform variant I. The instruction buffer at each computation element is capable of holding 16 instructions. Then we experiment with platform variant II maintaining identical size of instruction buffer. For all subsequent experiments we retain the platform variant II, since for the set of applications we are evaluating platform variant II performs better than variant I. In the next set of experiments, we change the size of instruction buffer available at the computation element to 32. In all of the experiments described so far, the instructions of the HyperOp were launched first followed by the data. Since the data is available only after all instructions are launched, this introduces significant delay between the availability of data at the orchestrator and the start of execution on the fabric. To reduce this delay in the next set of experiments we launch HyperOps by interleaving instructions and data corresponding to those instructions. This reduces the total execution time, as there can be more overlap between the execution of the HyperOp and the launching of the HyperOp. In the final set of experiments, we execute loops on the fabric using the end-to-end flow control mechanism described in section 6.1.2. These experiments are...
tabulated in table 7.3 for quick reference. As mentioned earlier, we perform these experiments on two sets of application. For applications from the cryptography domain, we use a homogeneous fabric containing the CPUs described in the table 7.1. The applications from linear algebra domain are mapped onto a heterogeneous fabric containing five floating-point units and twenty-five integer units.

Table 7.3: List of experiments

<table>
<thead>
<tr>
<th>Experiment Name</th>
<th>Orchestration configuration</th>
<th>CE instruction buffer</th>
<th>Launch of HyperOp</th>
<th>Loops on Fabric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config I</td>
<td>Platform Variant I</td>
<td>16 Slots</td>
<td>Instructions followed by data</td>
<td>No</td>
</tr>
<tr>
<td>Config II</td>
<td>Platform Variant II</td>
<td>16 Slots</td>
<td>Instructions followed by data</td>
<td>No</td>
</tr>
<tr>
<td>Config III</td>
<td>Platform Variant II</td>
<td>32 Slots</td>
<td>Instructions followed by data</td>
<td>No</td>
</tr>
<tr>
<td>Config IV</td>
<td>Platform Variant II</td>
<td>32 Slots</td>
<td>Instructions, data interleaved</td>
<td>No</td>
</tr>
<tr>
<td>Config V</td>
<td>Platform Variant II</td>
<td>32 Slots</td>
<td>Instructions, data interleaved</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The plot in the figure 7.8 shows the execution time of various configurations normalized with respect to configuration I. Configuration II performs better than configuration I in all applications. Configuration II gains primarily because of reduction in fabric execution time. This is due to the simpler pipeline in the computation element and hence shorter time to launch two dependent instructions. In some cases, it also gains because of the reduction in latencies introduced by the orchestrator. Since the orchestrator in variant II is simpler, it takes lesser number of cycles to check for the ready HyperOp once the data is delivered thereby reducing orchestrator overheads. However, the reduction in the execution time varies across applications depending on the application characteristics. In the following section, we group the applications based on these characteristics and discuss the effect on execution time of each group of applications.

- **CRC, Sobel, Matrix multiplication** have a similar load-store to compute ratio and all of them have small number of instructions. In addition, these applications do not exhibit a lot of parallelism. Since these applications are sequential, they primarily gain because of the reduction in fabric execution time. In configuration II, the delay between executions of two dependent operations is reduced to two cycles as opposed to four cycles in configuration I. The execution time of these applications
7.4 Analysis of execution time

Figure 7.8: Execution time of different platform variants normalized w.r.t. to Configuration I

improves by about 25% when compared to configuration I.

- In AES-E and AES-D applications we see a 10% reduction in execution time when compared to configuration I, as ILP can be better exploited due to the two cycle delay between dependent instructions as opposed to four cycles in configuration I. However, unlike the applications mentioned earlier AES-E and AES-D have a high ILP. However, not all of the parallelism is exploited, as we need to balance the communication latencies with that of gain obtained by exploiting parallelism. To reduce the communication delays few parallel operations are mapped to the same computation element. The ILP available within the computation element was used to hide the latencies between two dependent instructions. Hence, we see a 10% improvement as opposed to 25% improvement in the earlier applications.

- ECPA and ECPD applications gain the largest when compared to all other applications. The reason for this gain is the increase in number of inputs allowed per HyperOp in configuration II. In configuration I, where hardware controlled orchestration mechanism is used one input is reserved for storing the instance number of the HyperOp. This is not required in configuration II and hence an additional input can be accommodated. The number of inputs limits the HyperOps in these two applications and an additional input leads to larger HyperOps and reduction in the total number of HyperOps executed. This contributes
to the performance gains in addition to the reasons mentioned earlier. Hence, we see a 50% reduction in the execution time.

- MRI-Q, FFT and QR have several long latency operations like $\sin$, $\cos$ and $\sqrt{\cdot}$. These operations therefore dominate the execution time. Hence, we see relatively less improvement in these kernels.

- SHA-1, IDCT, and LU show a significant improvement in execution time. These kernels have relatively lesser parallelism when compared to AES-E and AES-D. Hence, we see a higher improvement in performance over AES-E and AES-D. These applications are larger than CRC and Sobel and spend a larger percentage of time on fabric. Hence, the performance gain is higher when compared to the earlier applications.

In configuration III, where we employ instruction buffer size of 32 as opposed to 16 in earlier configurations, we observe an interesting trend. Few applications have large positive impact and most applications either have negative or very less improvement. The large improvement is observed for large kernels viz. AES-D, AES-E, ECPA, ECPD. For these applications, the core kernel can be accommodated completely on the fabric in configuration III due to the availability of larger instruction buffers at each computation element. In the earlier configuration, this kernel had to be split into multiple HyperOps. The core kernel is a single basic block and the dataflow graph of the basic block needs to be split when it cannot be accommodated on the fabric. As mentioned earlier, the dataflow graph partitioning algorithm is greedy and maximizes the number of instructions in the first partition without considering the effect on the subsequent partitions. This led to a large number of outputs from the first partition and the subsequent partitions were limited by the number of inputs. Hence, the fabric could not be utilized efficiently for these partitions. Further, this also increases the instructions required for inter-HyperOp data transfers. All these reasons contributed to a large improvement in performance. In other applications the performance degradation can be attributed to

(i) the drop in parallelism exploited as they occupy lesser computation elements

(ii) increase in launch latencies due to the loss of parallelism while launching the HyperOp. Instructions and data corresponding to various partitions of a HyperOp are launched in parallel using peripheral routers. In our CGRA, five streams can be transferred in parallel. When we increase the buffer size to 32, lesser computation elements are required thus reducing the number of streams to be launched in parallel.

AES-E, AES-D, ECPA, ECPD and IDCT are large kernels and gain because of the increase in the capacity on the fabric. CRC, MRI-Q. Matrix multiplication,
7.4 Analysis of execution time

Sobel, FFT are small kernels and are not benefited by the increase in fabric capacity. Further, due to the above-mentioned reasons the performance degrades slightly. In QR and LU applications, the performance degrades significantly. This is due to the imperfect instruction scheduling algorithm used. The inter-HyperOp data transfer instructions are the leaf nodes of the DFG and the scheduling algorithm gives lesser priority to these instructions as expected by FDS. Due to this, the data delivery is delayed. In both QR and LU applications, the data that is to be transferred to the next iteration is produced in the beginning of the execution. However, the delivery of the data is delayed hence reducing the loop level parallelism available in these applications. The scheduling algorithm needs to be enhanced such that inter-HyperOp data transfers are scheduled depending on the parallelism available across the loops.

In Configuration IV, we launch the HyperOp by interleaving the data and instructions. The data corresponding to an instruction is launched along with the instruction. This optimization improves the performance of the applications by 5-18% over configuration III. A higher improvement is seen in applications that have lesser ILP. In such applications, the HyperOp launch delays could be masked by the execution time on the fabric due to the interleaving of instructions and data. In applications that have larger ILP, the launch delays could not be masked efficiently as the data was ready even before the instructions were available. Configuration V enables execution of loops on the fabric. Since we use a flow control mechanism based on end-to-end flow control, it introduces additional overhead between executions of two iterations of the loop. It also kills parallelism that is available across the loops. However, it reduces the launch latencies of the HyperOp as the instructions are launched once and executed multiple times. Thus depending on the parallelism available across loops, the size of the HyperOps and the overhead introduced due to flow control we see anywhere from 20% improvement to a minor degradation in performance. The analysis of these results is as follows:

- **CRC**, Matrix multiplication, FFT does not have parallelism across loop iterations and hence primarily gain because of the reduction in launch latencies. The overhead because of the flow control is proportional to the number of computation elements participating in the flow control. Since these kernels are small, the overhead is minimal. Hence, we see a higher performance improvement when compare to other kernels. AES-D, ECPA, and ECPD are larger applications and have a higher flow control overhead. Due to this, they show lesser performance improvement.

- **LU**, QR, and MRI-Q have loop level parallelism. Since the loop is executed sequentially, loop level parallelism cannot be exploited. This
loss in performance is balanced by the gain due to decreased launch latencies.

- In AES-E, we see a slight degradation in performance because of executing loop on the fabric. AES-E does not have loop level parallelism. It is a large kernel and the overhead due to the flow control mechanism is significant. Further, unlike AES-D, AES-E has parallelism towards the end of the iteration and is sequential in the beginning. Hence, it has no relative gain by eliminating launch latencies. Further, it incurs an overhead because of the flow control mechanism and hence we see degradation in performance.

- IDCT and Sobel have minor gains, primarily because of the reduction in launch latencies.

7.5 Pipeline Parallelism

As mentioned in section 6.2.1, exploiting pipeline parallelism is not fully automated in the compiler. In this section, we present the results of exploiting pipeline parallelism for streaming FFT kernel with the help of some manual intervention. The streaming FFT kernel was implemented as a pipeline, where each stage of the pipeline performs one stage of the FFT. Thus, the number of pipeline stages depends on the number of points. For a 1024-point FFT the pipeline consists of ten stages. This was identified manually and the information was provided to the compiler by specifying which HyperOps should be a part of each stage. The variables corresponding to the data communication occurring between the stages of FFT are manually annotated. Using the information provided, compiler generates necessary code for each of the pipeline stages including the instructions to send and receive data across these pipeline stages. Further, it is to be noted that the processing performed in each stage consists of a two-level nested loop. Compiler automatically transforms this loop to a well-behaved loop (as discussed in section 6.1.1) for execution on the fabric. Hence, a flow control mechanism is not required to execute the instructions within a stage. However, a flow-control mechanism is required across pipeline stages to ensure correct execution. Each stage of the pipeline is manually placed on the fabric, so that the flow control mechanism available at the routers is suitably exploited. This implementation of the FFT is for a fixed-point FFT and employs custom functional units to compute the butterfly operation. With such an implementation, a throughput of 2.3 Mega-sample/second at 100 MHz was obtained.
7.6 Comparison with other accelerators

In this section, we compare the performance obtained on our architecture with other accelerators. Since, there is no standard benchmarks in the domain of CGRAs and accelerators different teams use different set of applications to evaluate their work. Further, depending on the context of the work different parameters and different optimization criteria are used. This makes it difficult to compare different approaches. Hence, along with the comparison we also highlight some of the factors that help in analyzing the results. We compare with three different approaches viz. Molen polymorphic processor (Vassiliadis et al., 2001), ADRES (Mei et al., 2003a), Deadulus (Nikolov et al., 2008). We rely on the published results from these groups for comparison. We briefly describe the three approaches before we present the results.

- **Molen**: Molen is based on co-processor architecture paradigm that allows the use of reconfigurable custom function units. Molen has a general purpose processor connected with a reconfigurable fabric. The reconfigurable fabric can be customized to perform a particular operation. SET and EXECUTE instructions are introduced into the ISA of the processor to enable programming the fabric from the processor. SET instruction loads the configuration bit stream onto the fabric and EXECUTE instruction executes the instruction. All the experiments were conducted on a prototype platform implemented on a Xilinx Vertex board. In the prototype implementation, PowerPC running at 300MHz is treated as the general purpose processor and FPGA serves as the reconfigurable fabric. The custom functional units were hand-coded in VHDL.

- **ADRES**: ADRES is a CGRA that has a VLIW processor and a reconfigurable fabric associated with it. The reconfigurable fabric is a set of functional units interconnected using a point-to-point network. These set of functional units are treated as an extension of VLIW processor and the register file could be shared between the processor and the reconfigurable fabric. Loop kernels are compiled onto the reconfigurable fabric using modulo scheduling based techniques. All the functional units work in a lock-step and the compiler assumes deterministic latencies for all operations including memory reads and writes. The results reported in this section were obtained using a reconfigurable fabric of 4 × 4 ALUs.

- **Deadulus**: Deadulus is a system-level design tool for HW/SW co-designing. Deadulus takes a platform specification, application specification and a mapping specification as inputs. The application is specified using khan process network model of computation. In this model, application is specified as a set of parallel tasks and each task
can be mapped onto a unit in the platform. The parallel specification is obtained from a sequential C code using PNGen tool. The platform specification contains the components (could be custom IP blocks or processors) along with the interconnection. Mapping specification specifies how each application task is mapped onto a particular component on the platform. Deadulus generates the necessary hardware, software and code to synchronize between various units in the platform. The results reported in this section were obtained on a homogeneous platform containing five microblaze processors.

Figure 7.9 shows the comparison of our approach with that of other approaches. As mentioned earlier we could not obtain the results for all applications. The graph shows frequency normalized execution time for the applications for which results are available in one of the above mentioned approaches (Kuzmanov and Oijen, 2007; Chaves et al., 2005, 2008; Mei et al., 2004; Nikolov, 2009). In the case of Sobel edge detection, the results were available for a 128 x 128 image. These numbers are appropriately scaled for the image size of 382 x 204 which is the image size considered in our execution. We can see from the graph that the proposed approach out performs ADRES and Deadulus for the applications compared. In the case of Molen, the proposed approach performs slightly better in the case of AES-D and AES-E applications. However, we see a significant degradation in the case of matrix multiplication and SHA-1. In Molen platform custom functional units are employed for performing complete AES-Decryption, AES-Encryption, SHA-1 and Matrix multiplication. These custom functional units are developed manually at RTL level (coded in VHDL) and synthesized onto the FPGA. All these custom functional units are capable of performing the entire computation. In the case of matrix multiplication, they employed a custom functional unit that could multiply a block of matrix whose size is configurable. In the case of our approach, we do not employ custom functional units that are capable of performing entire computation. In the case of AES application, we employ custom functional units only for performing field multiplication of 32-bits. Thus, it is significant achievement to outperform a dedicated implementation. For the other two applications no domain customization was performed and hence degradation in performance. These results highlight the importance of domain customization to achieve good performance.
7.7 Comparison with GPPs

So far, we have presented the performance of the applications with respect to various configurations of CGRAs. In this section, we present the comparison with a General-purpose processor. We use Intel Core2 Quad Q8400 operating at 2.66GHz. Please note that though we are employing a Quad core, the applications were not written using threads and hence they execute on a single core. We use the same application specification to target both GPP and our platform. Most of the integer applications viz. AES-D, AES-E, ECPA, ECPD, IDCT, SHA-1, CRC and FFT do not contain thread-level parallelism. However, these applications have high degree of data level parallelism that can be exploited using SIMD extensions available in the general purpose processors. There are several implementations available in the literature Gueron (2012); Locktyukhin (2010); Hankerson et al. (2004) that use Streaming SIMD Extensions (SSE) to execute these applications efficiently. For example, an implementation of SHA-1 on Intel processor having SSE extensions, can execute SHA-1 operation in 672 cycles. However, all these implementations are coded in assembly with a detailed knowledge on the processor and the registers. As mentioned by Locktyukhin (2010) no compiler can do such an efficient job inspite of the maturity of the compilers for general purpose processors. We do not compare the execution time of our approach with that of implementations using SSE. However, these implementations point to the need of providing vector support on the proposed architecture and compiler to achieve very high performance. For the remaining applications threads can be employed to improve performance. However, it is to be noted that for all of the remaining applications using threads is beneficial only for applications

Figure 7.9: Execution time of our platform when compared with other accelerator approaches

![Graph showing comparison with GPPs]
working on large matrices. Further, the performance improvement obtained depends on various other factors like the cache size, cache structure etc. Also, various techniques (like blocking, transposing) are proposed for minimizing the effects of caches. In this thesis, we do not study these effects in detail. Further, threads can also be employed in the proposed architecture to obtain similar performance improvements. However, the platform is currently not mature enough to run these experiments. Hence, we compare the performance of these applications with that of single threaded performance. As mentioned already, both in the case of GPP and in the proposed architecture similar speed-ups are expected by employing threads. In our experiments we observe a performance improvement of 1.9X, 1.8X and 1.78X for matrix multiplication, LU factorization and QR factorization respectively when using two threads with a matrix size of $100 \times 100$. For a matrix size of $30 \times 30$, which is the size used in the current experiments, we actually see a slight performance degradation due to the overhead of thread creation and synchronization.

The performance comparison is presented in the figure 7.10. The figure presents a frequency normalized execution time for each of the kernels. The execution time is computed by multiplying the number of cycles with the frequency of operation. Our platform operates at 450MHz when synthesized using Faraday high-speed low $V_T$ 90nm technology library. Since the hardware implementation was not fine-tuned, we expect the frequency to go up to 700 MHz by redesigning few performance critical components. We present the comparison with GPP considering both 450MHz and 700MHz. We can clearly see two distinct trends in the performance when compared to a GPP.

![Figure 7.10: Execution time of our platform when compared with a General purpose processor](image)
7.7 Comparison with GPPs

The four cryptography applications, AES-D, AES-E, ECPA, and ECPD, perform an order of magnitude to two orders of magnitude better than a GPP. This improvement is primarily due to the following factors.

- We employ custom-made functional units for field multiplication and field squaring. These two operations involve bit-level operations and are highly inefficient when implemented using a normal ALU. Domain customization plays a very important role in improving performance.

- All these kernels show high degree of parallelism with very less control and hence benefit from a fabric that is built to exploit spatial parallelism.

- They have few load-store instructions, and hence we execute more often on the fabric. Accessing load-store units introduces considerable delays, as we have to go to the periphery of the fabric for every access.

For the rest of the applications, we see a significant drop in the performance when compared to a GPP. All these applications have lesser parallelism when compared to the earlier applications and have more load-store accesses. As mentioned earlier, accessing a LSU introduces a significant delay based on the location of the computation element. The best-case delay that can be achieved is 10 cycles, which is $10 \times$ more than the single cycle cache access in a GPP. Further, GPPs has features like branch prediction, bypass logic etc. that improves the performance of straight-line code significantly. Though the applications have some amount of ILP, we could not exploit it efficiently because of long latencies involved in communication between two computation elements. Further, dataflow execution inherently increases the amount of computation for control intensive applications because of the additional nodes introduced to capture the control and notion of order present in the imperative specification. In summary, the performance degradation is attributed to the following reasons,

- Long latency load-store accesses

- Inefficiency in exploiting the ILP because of latencies incurred due to communication between computation elements

- Additional operations introduced to capture the control information of the imperative specification

- Lack of bypass logic, which introduces two-cycle latency between the execution of two dependent instructions

- Floating point functional units are not pipelined and are implemented as multi-cycle operations. Since these are long latency operations, these introduce additional delays for floating point applications.
Based on the earlier experiments we can observe that the proposed platform is good for applications that have large amount of parallelism and have loop-level parallelism between kernels that execute for considerable amount of time on fabric. Put in another way, if the outer level of loop has loop-level parallelism then it can be exploited more efficiently. Further, we have used only end-to-end flow control mechanism. The end-to-end flow control mechanism introduces delay between the execution of two iterations of the loop. This delay can be reduced by transforming the loops to well-behaved loops (discussed in section 6.1.1). However, we could not evaluate loop-transformation technique because of the lack of support in the hardware.

7.8 Performance Enhancements

To validate the observations, we implement few enhancements in a simulator and estimate the performance improvement because of these enhancements. We implemented a low latency load-store access and introduced bypass logic. The presence of bypass logic enables two dependent operations to execute in consecutive cycles. We have chosen these two enhancements, as they can be implemented without affecting the area and power significantly. Further, these enhancements were not implemented in the synthesizable version of the platform. The results presented earlier were from synthesizable hardware. Since both these enhancements are related to the fabric execution only, we implemented a simple cycle accurate simulator for the fabric. Hence, we do not execute the entire application. We execute one HyperOp at a time because of the limitations of the simulator. We execute important HyperOps in each of these applications. Results of this experiment are shown in figure 7.11 for cryptography kernels and figure 7.12 for floating point kernels. This plot shows the performance improvement because of the above mentioned enhancements. CRC, SHA1-4, SHA1-5 HyperOps see a significant improvement as they benefit from both these enhancements. All these HyperOps contain load-store operations and are sequential in nature. AES, ECPA and IDCT have lesser number of load-store instructions in comparison to the number of compute instructions. Further, these applications also exhibit parallelism and hence we see a lesser impact of these optimizations on these kernels. In all of these applications, the performance improvement obtained is directly proportional to the percentage of load-store instructions and is inversely proportional to the amount of parallelism available in the application. The only exception is HyperOp ECPA-2, which gains significantly. This is due to FDS scheduling algorithm employed for scheduling operations. ECPA-2 contains 13 load instructions and the output of these loads is transferred to other HyperOps. It does not perform any other computation. The load and the dependent transfer instruction are scheduled in two consecutive slots. Hence, all the operations become sequential and incur round-trip delay for each of
7.8 Performance Enhancements

the load-store instructions. Hence, we see a significant improvement due to the reduced load-store latencies. However, in an alternate schedule where loads are scheduled ahead of data transfer instructions, we would not have observed this anomaly. Matrix multiplication-3 and LU-3 are small HyperOps

![Graph showing Normalized Fabric Execution Time w.r.t. configuration III](image)

**Figure 7.11:** Execution time of our platform when compared with a General purpose processor and the floating-point computation dominates the execution. Hence, we see relatively lesser improvement in these kernels. FFT-2 calculates the twiddle factors using sin, cos functions. These two functions dominate the execution time. Hence, we see only a minor improvement in performance. We can see that, for most of the floating-point kernels, which were not doing better than the general-purpose processor we see an improvement of close to 50%. Thus, the inefficiency of the platform for kernels for linear algebra domain is due to the lack of fine-tuning the reconfigurable fabric.
Figure 7.12: Execution time of our platform when compared with a General purpose processor

7.9 Summary

In this chapter, we executed two sets of applications from different domains to show the flexibility of the compiler. The applications were chosen from cryptography domain and linear algebra domain. They were targeted to a platform consisting of $5 \times 6$ computation elements and interconnected using a lightweight NoC. For applications from cryptography domain, a custom functional unit was provided to accelerate field multiplication and field squaring which are primitive operations in many cryptographic standards. In cryptography domain apart for domain customization, we could also exploit parallelism available in those applications efficiently. For linear algebra domain, we use five floating-point units. We perform experiments with various platform configurations and provide the analysis of the results. We could achieve low reconfiguration overhead because of the presence of NoC and the dataflow execution paradigm. In applications from linear algebra domain, we could efficiently exploit loop-level parallelism. The reason for this degradation can be attributed to the use of non-pipelined floating point units, and the delays involved in accessing memory. We compared the performance of the CGRA against an Intel Core2 Quad processor. The cryptography applications performed more than an order of magnitude better than the General purpose processor, whereas, the linear algebra kernels performed close to an order of magnitude worse than the GPP.
Chapter 8

Conclusion and Future work

In this section, we present the summary of the thesis and briefly list the contributions of this thesis. We then present few important observations we have made when executing the applications on our CGRA. We then present the future work based on these observations.

8.1 Summary

In this thesis, we have presented a compiler for CGRAs employing dataflow based execution paradigm. Dataflow execution paradigm has inherent support for exploiting lightweight parallelism that exists across loops and function calls. Further, dataflow execution can also schedule the application partitions efficiently without host processor intervention. This allows us to execute entire applications as opposed to application kernels supported by many CGRAs. One of the main objectives of this work is to ease the programming for CGRA. The compiler supports full C89 standard and can compile it to the target architecture without manual intervention. We accepted user hints/intrinsics for performance critical portions to achieve higher efficiency. The major part of this thesis studied the correctness aspects involved in orchestrating these partitions. The interactions between the various partitions have a well-defined interface and the compiler is oblivious to the processing happening within a partition. This allows us to support different execution fabrics for a partition and the orchestration of partitions is ensured to be correct even in the presence of different kinds of fabrics. Further, we also assume fabric to have a capability to wait for the data before executing the operation. This feature allows us to implement pipeline parallelism efficiently. It also helps in tolerating variable latencies in the presence of memory hierarchy or data dependent execution. Further, this also helped in reducing the configuration latencies. The compiler was built to efficiently exploit different
Conclusion and Future work

kinds of parallelism. The various steps in the compilation trajectory are as follows

- The first step of the compilation transforms the imperative specification into a dataflow graph. Control information in the imperative specification was transformed to dataflow by employing predicates. Precedence edges were employed to capture the notion of order that is inherent in the imperative specification. Precedence edges were added only between operations affecting global state. Alias information was used to restrict these precedence edges to the memory operations that may alias. These edges are not required between memory operations that are known to point to different locations. Predicated steer nodes were employed to deliver data at merge points. These were also employed to deliver data for intra HyperOp communication. ROBDD based optimization was proposed to build efficient dataflow graphs by eliminating redundant operations and edges. The redundancy might arise due to (i) the predicated steer nodes employed for delivering data at merge points (ii) memory precedence edges added (iii) predicate edges for transforming the control information.

- The next step partitions the dataflow graph into multiple partitions called HyperOps. We derived the set of constraints that have to be satisfied for correct execution of HyperOps. These constraints include (i) HyperOps are vertex induced subset of the dataflow graph (ii) HyperOps are disjoint, i.e. every operation belongs to at most one HyperOp (iii) All the operations in the dataflow graph are assigned to a HyperOp (iv) HyperOps satisfy convexity condition i.e. there are no cycles in the HyperOp Interaction graph. Apart from these correctness constraints, HyperOps also should obey structural constraints based on the capabilities of the execution fabric and the orchestrator. These constraints include (i) limit on the total number of operations allowed in a HyperOp (ii) The total number of inputs allowed for a HyperOp. We proposed an algorithm that uses both dataflow and control flow information to form HyperOps. The control information was used to derive efficient checks to satisfy the correctness constraints mentioned earlier. The dataflow information is used to check for structural constraints.

- The communication between HyperOps is facilitated by the orchestrator. The compiler supports two variants of the orchestrator.
  
  - In first variant, compiler manages the resources at the orchestrator. Compiler requests for a specified number of locations to be allocated at the beginning of the function, and all the HyperOps that belong to his function use the same set of context memory locations. The objective of the compiler is to minimize the number
of context memory locations required. We model this problem similar to that of register allocation in traditional compilation. We define the def and use of a HyperOp in order to transform this problem to that of register allocation and compute the liveness of each HyperOp.

– In the second variant, hardware is responsible for managing the resources on the orchestrator. In this scheme, hardware is responsible for allocating a new context memory location when data is available for a new instance of a HyperOp. Instance numbers were employed to distinguish between various instances of the same HyperOp. The instance number of the consumer was computed using the instance number of the producer. We discussed an efficient scheme to compute instance number of consumer based on the hints provided by the compiler.

In both these variants, additional processing is required to handle loop-invariant data, and to clear the data belonging to HyperOps on not-taken paths. We also described mechanisms to handle the same.

• In the last step, each HyperOp is partitioned further and mapped onto the execution fabric. This step requires detailed knowledge of the capabilities and limitations of the fabric. This step is not in the scope of this thesis.

The HyperOps that are formed in the first pass do not contain loops. We did not include loops in HyperOp since the reconfigurable fabric is not assumed to have support for flow control. However, in certain situations executing loops on the fabric helps achieve the required performance. Hence, we introduced Loop-HyperOps that can contain loops on the fabric. To be able to do so, we have proposed a loop transformation that allows the loops to be executed on a fabric that lacks flow control. The basic idea behind this technique is to ensure that all operations in a loop have to be executed before restarting the next iteration. This technique is not capable of handling multiple levels of nesting since delivering loop-invariant data requires additional processing. Unlike other techniques proposed earlier in the literature (Mei et al. (2003a), Park et al. (2009)), this technique works even in the presence of variable latencies. To extend the support to execution of multiple levels of nesting we presented an end-to-end flow control scheme by adding minimal support in the hardware. We also discussed various issues involved in using near-neighbor flow control mechanism for achieving pipeline parallelism. The compiler is also capable of accepting intrinsics/user hints to fine-tune the generated code for performance critical portions. The user hints could be in terms of specifying accurate alias information to expose more memory parallelism, specifying parallelism across function calls etc.
8.2 Contributions

The main contribution of this thesis is the compiler that is capable of accepting the full C89 standard and that can be targeted to a family of CGRAs employing dataflow execution paradigm. This eases the design space exploration and provides a unified compiler for several variants of the CGRA. Though the compiler currently does not integrate several traditional analyses like polyhedral analysis, it is built to accept the output of these compilation passes. For instance, it can accept the information about parallelism across function calls and this information can be used to replace user hints that can be provided to exploit TLP. In future, such analysis can be integrated to automate the compiler further. We have discussed various aspects to ensure correctness, when orchestrating the application partitions and described techniques to ensure the same. More specifically the contributions include

- ROBDD based optimization to obtain efficient dataflow graphs from an imperative specification.
- We derived the set of necessary and sufficient conditions for the correct execution of application partitions (HyperOps). Then we proposed an algorithm to partition the application that uses both control flow and dataflow information. This algorithm partitions the application honoring the conditions we derived earlier.
- We discussed various issues that arise when managing the context memory in a distributed dataflow execution paradigm and proposed mechanisms to handle each of these. These include (i) a mechanism to deliver the data to correct instance of the HyperOp (ii) a mechanism to purge data for HyperOps on not-taken paths (iii) an algorithm to allocate context memory efficiently to HyperOps of an application.
- We proposed a loop transformation technique to execute the innermost loop on a fabric that lacks support for flow control. We also described mechanisms to execute nested loops on the fabric. Both of these techniques are designed to work even in the presence of non-deterministic latencies.

8.3 Observations

In the process of compiling and executing applications, we made several observations relating to the performance of these applications. These observations provide directions for enhancing the compiler.

- Our initial attempt at partitioning the application considered the dataflow graph alone. The control information is not evident in the dataflow
8.3 Observations

graph. Ignoring the control flow information leads to sub-optimal partitions. The basicblocks typically contain closely related instructions, and it is more efficient to include all of these in a single HyperOp. Further, the use of control information allows us to design efficient checks for the correctness conditions. Another important observation related to the partitioning is that it is more efficient to have single entry HyperOps as opposed to multiple entry HyperOps. Multiple-entry HyperOps require more bookkeeping information to orchestrate the HyperOps correctly. Further, in the presence of multiple entry points, it is known that a set of instructions are not required to be executed even before launching the HyperOp. It is an additional overhead to launch these instructions and squash them subsequently. Thus, it is more efficient to group them into different HyperOps.

• We employed a NoC on the reconfigurable fabric to connect various computation elements. Each computation element was capable of issuing a single instruction every cycle. When a NoC is employed to connect computation elements on the fabric, compiler is not responsible for choosing paths and hence is less constrained when scheduling the operations. This choice is in tune with the objective of programmability. However, the use of NoC for small computation elements proved to be a bottleneck while exploiting ILP. The NoC introduces a latency of 3-4 cycles between near neighbor communication and hence it is not efficient to exploit fine-grained ILP because of the communication latencies. For applications like AES, ECPA and ECPD that have an ILP of 6-8, it does not prove to be a bottleneck. For the rest of the applications, this limits the performance that can be obtained. Thus, the choice of the interconnection network and the granularity of the computation element have to be chosen carefully depending on the application characteristics.

• Application-aware compilation is essential to obtain high performance. The compilation has to be aware of application characteristics and the capabilities of the CGRA in every phase of the compilation. We list few of these aspects here.

  – When creating the dataflow graph, it is more efficient to employ predicated steer nodes to deliver data when the producer and consumer are not separated by many levels of nesting. This reduces the number of purge instructions that have to be added.

  – When exploiting memory parallelism, we need to estimate the potential benefit that can be obtained by exploiting the parallelism and the amount of synchronization required due to the parallelism. In some cases, we observed that the synchronization costs outweigh the benefits gained from exploiting parallelism. This is
especially true when all the memory operations are required to be serviced at a single memory bank. Since the LSU services the requests in sequence, it is not advantageous to exploit parallelism on the fabric and pay a price for additional synchronization.

- Loop level parallelism has to be considered when scheduling the inter HyperOp data transfer instructions. The schedule of these instructions determines the amount of loop level parallelism exposed to the hardware. These instructions are the leaves in the dataflow graph and if special care is not taken, most scheduling algorithms give them least priority and schedule them at an available slot.

8.4 Future work

In this thesis we have presented a compiler for compiling applications onto employing dataflow execution paradigm. In the process of developing and using this compiler, we have made several interesting observations. In this section, we present the ideas that can be explored further to enhance the compiler based on the above stated observations.

8.4.1 Performance enhancements

The focus of this thesis has been the correctness aspects of the compilation process. We list few important optimizations here. There are several more optimizations, which can be applied to achieve higher performance.

- We need to revisit the traditional optimizations and measure the effect of these optimizations on the performance in the context of All traditional optimizations might not be suitable in the context of For example, consider the Loop-Invariant Code Motion (LICM) optimization. In the context of an aggressive LICM might lead to an increase in the number of inputs. For HyperOps that are limited by the number of inputs, this further restricts the size of the HyperOp. The reduction in the size of a HyperOp brings down the performance significantly. Thus, the performance gained by eliminating the re-computation is offset by reduction in the size of the HyperOp size. Another example is loop-un-switching optimization that moves a conditional present inside a loop to outside of it by duplicating the loop body. In the context of for loops that do not have loop-level parallelism, it is more efficient to launch them onto the fabric once and execute them repeatedly. Loop-un-switching makes such optimizations difficult to implement.
8.4 Future work

- Application and throughput-aware compilation is essential to achieve good performance and energy efficiency. Compiler needs to be enhanced to use an analytical model to estimate the effect of various optimizations performed. For example, consider the LICM optimization discussed earlier. This optimization improves performance as long as the increase in the number of inputs does not negate the benefit obtained by LICM. Compiler needs to evaluate how many computations can be moved out of the loop to obtain an efficient implementation. Another example is when exposing memory parallelism available in the application. Higher memory parallelism will lead to a higher number of synchronization nodes. Further, when hardware cannot service multiple memory operations in parallel then exposing memory parallelism might lead to degradation in performance. The use of analytical model to estimate the execution time will help the compiler to choose the right set of optimizations.

- We need to employ techniques for dependency analysis to automate the task of exploiting pipeline parallelism. Polyhedral analysis is an efficient technique for analyzing the dependencies (Pouchet et al., 2008; Bondhugula et al., 2008; Bastoul, 2004). Most of the embedded kernels fall under the class of affine control loops, which can be analyzed using polyhedral analysis. There are several frameworks like PolyLib (Clauss and Loechner, 1996), Graphite (Trifunovic et al., 2010), Polly (Grosser et al., 2011b), isl (Verdoolaege, 2010) that support polyhedral model. The dependency information obtained by employing these techniques needs to be used by the compiler to determine the optimal number of pipeline stages depending on the throughput requirements. The dependency information also helps in determining the data that can be streamed to various stages of pipeline. For loops that do not fall in the category of affine nested loops and are difficult to analyze, compiler should be capable of accepting user directives about dependency information. Similar to openMP/openHMPP to specify parallelism for a shared memory multiprocessor a standard can be defined to specify fine-grained parallelism, pipeline parallelism, streaming the data etc.,

- Techniques for inter-procedural analysis have to be employed to detect task level parallelism available across function calls. Similar to the dependency analysis compiler should be capable of taking user directives when necessary.

- In this thesis we have presented the use of imperative languages to program the CGRA. We have also suggested that there is a need to enhance the application specification through use of compiler hints/pragmas to explicitly identify regions of parallelism available in the application. Another possibility is to use a different class of languages like
Conclusion and Future work

functional languages. The use of functional languages poses different kind of challenges such as reclaiming the data memory and containing the parallelism etc. This is in contrast to the challenges faced when using imperative languages. When using imperative languages, the challenge lies in exposing parallelism by performing accurate analysis to disambiguate memory references, to identify dependencies across functions etc. Functional languages are suited for applications that are highly parallel. Thus, it will be interesting to study the use of functional languages to target CGRAs.

8.4.2 Just-in-time Compilation

Another important enhancement would be to provide just-in-time compiler support to achieve binary compatibility. Currently, the HyperOps are formed based on the parameters specified at compile time. At runtime if a larger fabric is available, then we will not be able to utilize the full potential of the fabric, as it requires a recompilation with new parameters. However, if we have JIT compiler that is capable of combining multiple HyperOps on the fly with minimal overhead then we can achieve binary compatibility. This also helps in choosing the right HyperOps depending on the resource availability on the fabric when multiple applications are competing for the same fabric. Depending on the throughput requirements, a run-time system can specify the resources allocated to each application and the application can scale at runtime for the specified number of resources.
References


REFERENCES


Publications


application-architecture design space exploration.” In “Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems,” CASES ’09.  


Satrawala, A. N., Varadarajan, K., Alle, M., Nandy, S. K., and Narayan, R. (2007). “REDEFINE: Architecture of a SOC Fabric for Runtime Composition of Computation Structures.” In “FPL ’07: Proceedings of the International Conference on Field Programmable Logic and Applications,” contribution:This is a paper that describes the philosophy of our work. This was the foundation for our work. All authors share equal contribution in this paper.